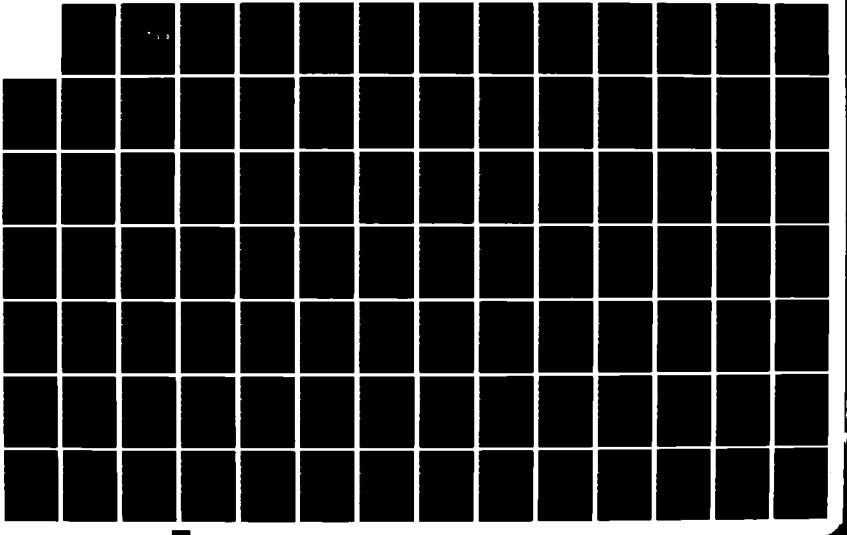
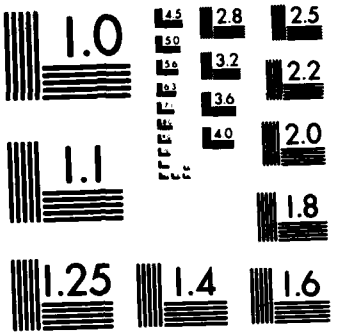
AD-A154 438 COMPUTER PERFORMANCE MODELING TOOL (CPMT)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA K A PAGEL DEC 84 1/2

UNCLASSIFIED F/G 9/2 NL





NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPUTER PERFORMANCE
MODELING TOOL (CPMT)

by

Karen A. Pagel

December 1984

Thesis Advisor: Alan A. Ross

Approved for public release; distribution unlimited

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Computer Performance Modeling Tool (CPMT)		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December 1984
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Karen A. Pagel		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943		12. REPORT DATE December 1984
		13. NUMBER OF PAGES 172
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Simulation
Discrete Event Simulation Program
PASCAL Simulation Program

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Computer Performance Modeling Tool (CPMT) is a discrete event simulation program designed to model computer systems. It is written in PASCAL for the VAX-11/VMS environment. CPMT uses the concepts of queueing theory to model computers as a network of server groups through which job events are processed. The interactive program provides the user the capability to update an indexed sequential data base of simulation model

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

specifications and to execute simulation runs of computer system models contained in the data base. Simulation model runs produce output statistics on the performance of the modeled computer system. The thesis documentation includes a User's Manual; information on computer system model design; CPMT data base and program specifications; program test and verification results; and enhancement possibilities to be included in the ongoing CPMT development project.

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A/1	

Approved for public release; distribution unlimited.

Computer Performance Modeling Tool (CPMT)

by

Karen A. Pagel
Lieutenant, United States Navy
B.A., Yale University, 1978

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1984

Author: Karen A. Pagel

Approved by: Alan A. Ross, Thesis Advisor

Norman R. Lyons, Second Reader

Willis R. Greer, Jr., Chairman,
Department of Administrative Sciences

Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

The Computer Performance Modeling Tool (CPMT) is a discrete event simulation program designed to model computer systems. It is written in PASCAL for the VAX-11/VMS environment. CPMT uses the concepts of queueing theory to model computers as a network of server groups through which job events are processed. The interactive program provides the user the capability to update an indexed sequential data base of simulation model sepcifications and to execute simulation runs of computer system models contained in the data base. Simulation model runs produce output statistics on the performance of the modeled computer system. The thesis documentation includes a User's Manual; information on computer system model design; CPMT data base and program specifications; program test and verification results; and enhancement possibilities to be included in the ongoing CPMT development project.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

The Computer Performance Modeling Tool (CPMT) is a discrete event simulation program designed to model computer systems. It is written in PASCAL for the VAX-11/VMS environment.

A. PROJECT PURPOSE

CPMT program development began as a class project for Computer Performance Evaluation, CS4400, taught at the Naval Postgraduate School. The intent of the project was twofold: first, to have students gain familiarity with the concepts of simulation programs through the process of program design and implementation; and second, to produce a viable simulation program which could be used within the class context to model computer systems and perform statistical analysis of the simulation model results. The class effort produced the initial simulation program design concept and the basic versions of two program modules.

B. SCOPE OF EFFORT

This thesis is a continuation of the CPMT program development task, with the goal of producing an operational, documented and tested baseline simulation program to be used as a classroom tool for CS4400 and as a basis for further student program development and enhancement projects. The thesis effort included adding interactive 'user friendly' features to the program; rewriting the main program Execute and Tabulate module; writing User's Manuals and system documentation, and testing the validity of the simulation program results.

C. OVERVIEW OF THE CPMT PROGRAM

CPMT uses the concepts of queueing theory to model computers as a network of server groups through which job events are processed. The user provides parameters to model computer systems in terms of both the computer system configuration and the job types run on the system. The computer configuration is modeled as a collection of server groups which represent system components such as CPU or disk drives. Additionally, an entrance and exit server group is specified to route jobs into and out of the system. Job types are modeled from parameters which define the job arrival rate and distribution; job priority; job routing probabilities from server group to server group; and the job service rates at the server groups.

After modeling the computer system and entering the model parameters in the CPMT data base, the user can execute the simulation model to produce output statistics concerning characteristics of the modeled computer system. Output statistics include items such as job type response times and utilization rates of system components. At simulation run time, the CPMT program generates jobs from the job type parameters and processes the jobs through the server group structure which represents the modeled computer. As the program processes the jobs it gathers data concerning the jobs and server groups. Upon completion of the simulation run the program tabulates statistical output from the gathered data.

CPMT is an interactive program comprised of four main modules under the control of the CPMT program driver. The four main modules are: the Update Module which provides an interactive capability to update the simulation model data base; the Check Simulation Specifications module which provides a capability to check the model parameter specifications for completeness and consistency; the Create Job

Stream module which creates the jobs to be run through the system from the job type parameters; and the Execute and Tabulate module which processes jobs through the server group structure and produces the statistical output.

D. THESIS ORGANIZATION

Chapters 2 and 3 of the thesis are user oriented. Chapter 2, Designing the Simulation Model, concerns the model design process. It describes the model specification parameters and their organization into job type, routing and server group records; discusses the design requirements and limitations; and provides an example of a model design. Chapter 3, the CPMT User's Manual, describes how the CPMT program is run. It includes descriptions of the online options for updating the data base and running the simulation model. As suggested by the ordering of Chapters 2 and 3, model design is best accomplished as a paper process that the user completes before using the CPMT program online. The user will probably find it helpful to complete the model data forms provided in Figures 2.6 and 2.7 during the model design process. The forms organize the model specification parameters into a format which facilitates online data entry.

Chapters 4 and 5 are oriented towards programmers concerned with CPMT maintenance and enhancement. Chapter 4, the Program Specifications, presents an overview of the CPMT driver and main modules. Chapter 4 also contains a data dictionary describing the dynamic data record structures and data items used by the CPMT program and a discussion of the physical files which comprise the system. Chapter 5, the Data Base Specifications, describes the indexed sequential data base.

The test an validation results for the CPMT program are discussed in C .pter 6. The conclusions of Chapter 7 present a list of possible program enhancements for continued program development.

II. DESIGNING THE SIMULATION MODEL

The most difficult aspect of using the Computer Performance Modeling Tool is designing the computer system model that is to be simulated. The utility of the simulation results will depend on the quality of the model design and the proficiency of the user in isolating the characteristics of the computer system components which have the greatest impact on system performance. This chapter is concerned with the development of the model specifications and a discussion of the input options and parameters which the user has available in the modeling process.

The design of the computer system entails two aspects: modeling the computer configuration and modeling the workload which is processed by the computer. The data parameters used to model the computer system are grouped into three record types for data input and data base storage: the job type records, the routing records, and the server group records. The job type and routing records describe the work to be performed by the system, and the server group record describes the attributes of the computer being simulated.

A. DESCRIPTION OF INPUT PARAMETERS

In this section, the design of the computer system model is discussed in terms of the data fields for the server group, job type and routing records. First, the Simulation Model Number, which is a common to the three data records, is described.

- **Simulation Model Number:** Each model is assigned a simulation model number between 1 and 99. The simulation number is used to identify a particular simulation model

in the simulation model data base. The simulation model number is common to all the record types developed to describe a given model.

1. Server Group Record

The computer is modeled as a collection of server groups. Each server group is comprised of one or many servers and is serviced by a single queue. The maximum queue length for each server group is assumed to be infinite. Server groups may be used to model components of the computer such as terminals, printers, I/O channels, CPU, disks. For each simulation model, the single server group record lists the server groups of the model.

Server Group Record Data Fields.

The server group record data parameters are discussed below and listed in Figure 2.1.

- **Server Group Number.** Currently CPMT is set up to accommodate 9 "working" server groups. The user assigns one of the available server group numbers 1 through 9 to the server groups in their model.
- **Number of Servers.** For each of the working server groups 1 through 9, the user identifies the number of servers in that server group. Valid range for the number of servers in the server group is 0 to 999. If a server group is not used in the user's model, then the user should specify '0' as the number of servers for that server group.

It is important to keep in mind that if a server group is identified as having several servers, the servers must be interchangeable since the assignment of servers to a

Record Field	Type	Range	Comments
Simulation Model Number	I	1..99	
Number Servers	Array 1..10 of Integer	1..999	

Figure 2.1 Server Group Record Parameters.

job within a server group is arbitrary. For example, suppose a computer system has two disks. If the jobs being modeled to 'run' on the system must access a disk, but not necessarily a particular disk, then the user may choose to model the system using a 'disk' server group having two identical servers. However, if the jobs must access a particular disk, then the user may wish to model the system as having two disk server groups, each with a single server.

In order to facilitate the entrance and exit of jobs into the system, entrance and exit "dummy" server groups are identified. A job always enters the system at Server Group 0 and exits the system at the highest number Server Group that the system is set up to handle, which is currently Server Group 10. No processing of job events takes place at either the entrance or exit server groups so there are no servers at either Server Group 0 or 10. The entrance and exit server groups exist as a routing mechanism and are further discussed in the section on routing records.

2. Job Type Record

The user models the work that is done by the computer as job types. Jobs with different processing characteristics are categorized into different job types. For

example, the user may wish to define a computer system which has two basic job types: jobs that are I/O intensive, and jobs which are CPU intensive.

The simulation model can accommodate from 1 to 99 different job types. Each job type is described with a job type record and multiple routing records which are subordinate to the job type record.

Job Type Record Data Fields. The job type record data parameters include: job type number, job type arrival distribution, arrival distribution parameter, and job type priority. The record fields are discussed below and listed in Figure 2.2.

- **Job Type Number:** Each job type is assigned an integer value from 1 to 99 for purposes of identification. The user should be sure to assign sequential numbers to the job types commencing with 1 when designing the model. The reason for this is that the data update module used for input of the data base specifications automatically assigns job type numbers as the job type records are entered. The user needs to enter the job type record data in the order corresponding to job type numbers assigned in the model design process.

- **Arrival Distribution and Distribution Parameter.** In order to describe the job type arrival rate into the system the user selects a distribution type and a 'distribution parameter' which depends on the distribution type selected. The distribution and distribution parameter options are discussed in section B of this chapter.

- **Job Type Priority.** For each job type, the user specifies the priority which that job will have in the system. The priority range is from 1 to 10, with 1

being the highest. Jobs events will be serviced at the server groups based on an ordering of jobs by queueing discipline within priority. For example, if the queueing discipline at a given server group is first come, first served, then all the jobs of priority 1 will be processed according to their arrival time before the processing of jobs of priority 2.

Record Field	Type	Range	Comments
Simulation Model Number	Integer	1..99	
Job Type Number	Integer	1..99	
Arrival Distribution	Integer	1..3	1 - Deterministic 2 - Exponential 3 - Uniform
Arrival Distribution Parameter	Integer	1..99999	See Figure 2.4
Job Type Priority	Integer	1..10	

Figure 2.2 Job Type Record Parameters.

3. Routing Record

The routing record has two functions: it describes the service rate of job events of the given job type at the server group and it describes the routing probability for the job type from the server group to all the other server groups in the system.

Routing records are subordinate to the job type records. Each job type record requires from 2 to 10 associated routing records. A routing record is required for the entrance server group and for each server group that the job can potentially visit in its progression through the computer system, excluding the exit server group.

Routing Record Data Fields. Routing record data fields are described below and listed in Figure 2.3.

- **Server Group Number.** The routing record server group number is identified with an integer in the range of 0 to 9 corresponding to server groups 0 through 9.
- **Service Distribution and Distribution Parameter.** The service rate of the job type is defined in terms of distribution type and a corresponding distribution parameter. See section B of this chapter for the discussion of the distribution parameter options.
- **Routing Probability.** The routing probability is implemented as a one dimensional array of integers. The array index corresponds to server group numbers 1 through 10. The user enters the percent probability that the job will go from the server group which is the subject of the routing record to the other server groups in the system. The routing probability is an integer value from 0 to 100. The total of all routing probabilities from a given server group must equal 100.

Routing Design Guidelines. The routing design for each job type must meet the following requirements:

- A routing record is required for Server Group 0, the entrance server group. It is necessary to provide the routing probability data for SG 0, since it will define the entrance of the job into the system. However, since

Record Field	Type	Range	Comments
Simulation Model Number	Integer	1..99	
Job Type Number	Integer	1..99	
Server Group Number	Integer	0..9	
Service Distribution	Integer	1..3	1 - Deterministic 2 - Exponential 3 - Uniform
Service Distribution Parameter	Integer	0.. 99999	See Figure 2.4
Routing Probability	Array 1..10 of Integer	0..100	

Figure 2.3 Routing Record Parameters.

no processing is done at the entrance server group, no values are assigned to service distribution and distribution parameters for Server Group 0.

- Jobs are never routed to SG 0, the entrance server group.
- Jobs must be routed to SG 10, the exit server group, from at least one other server group in the system.
- No routing record is required for SG 10 because no processing is done at the exit server group and because the job is not routed to another server group from SG 10.

- The sum of the routing probabilities from a given server group to server groups 1 through 10 must equal 100.
- The probability of routing a job from a given server group to itself must not equal 100, to avoid generating a job which will never complete processing.
- If a job is routed to a server group, then a routing record must exist for that server group to provide for routing the job from that server group.

B. DISTRIBUTION PARAMETERS

To describe the arrival rates and service rates of job types, the user selects one of three available distribution types and supplies the requisite distribution parameter for the distribution type selected. The three distribution types currently implemented in the CPMT are the deterministic, exponential, and uniform distributions. Figure 2.4 lists the available distribution types and corresponding distribution parameters.

DIST CODE	DIST TYPE	DISTRIBUTION PARAMETER
1	Deterministic	Deterministic Value
2	Exponential	Exponential Distribution Mean
3	Uniform	Upper Bound X of Uniform Distribution from 0 to X

Figure 2.4 Distribution Types and Parameters.

1. Deterministic Distribution

In the deterministic distribution, the servicing time or interarrival time of the jobs is a given value. When selecting the deterministic distribution, the user specifies a parameter which is the given time unit of the service duration or interarrival rate. For example, if a given message always requires two time units of processing time by the CPU, then the user would specify the deterministic distribution and select '2' as the distribution parameter when modeling that portion of the job type.

2. Exponential Distribution

When selecting the exponential distribution to characterize job service and arrival rates, the user specifies the mean of the distribution as the distribution parameter.

3. Uniform Distribution

For a uniform distribution, the distribution is uniform over the range 0 to X, where X is the upper bound of the uniform distribution range. The user specifies the upper bound of the range when selecting the uniform distribution.

C. GENERIC TIME UNIT

CPMT is implemented with a "generic" time unit. The internal "clock" of the execute and tabulate module of the CPMT which runs the simulation is implemented as an integer. Thus all service times and arrival times must be represented as integer values. It is up to the designer of the simulation model to determine what this time unit represents when specifying service and arrival rates for the job types in the model. The time unit must remain consistent throughout the simulation model if the statistical results are to be

consistent. The time unit representation selected should be the smallest unit required to specify model parameters since integer values are used and times cannot be represented as a fraction of a time unit. The selection of a time unit is necessary only for model design and correct interpretation of the statistical results of the simulation. The time unit selected is not entered into the program nor used in simulation execution.

D. MODEL DESIGN FORMS

Several forms are provided to aide the user in the model design process and preparation of model specification input data.

1. Job Type Routing Diagram Form

The Job Type Routing Diagram Form is provided in Figure 2.5. The user may find it helpful to diagram the job routing for each job type in the simulation model. The routing is diagramed by drawing routing probability vectors between the server groups and labeling them with the routing probability. Space is also provided on the form to indicate the service type distribution and service parameter for the service rates of the job type at each of the server groups. An example of a prepared Job Type Routing Diagram Form is presented in Figure 2.10. The diagram of the job type routing provides a visual display from which the user can check the job type routing to ensure that it meets the guidelines discussed in the Routing Record subsection of section A of this chapter.

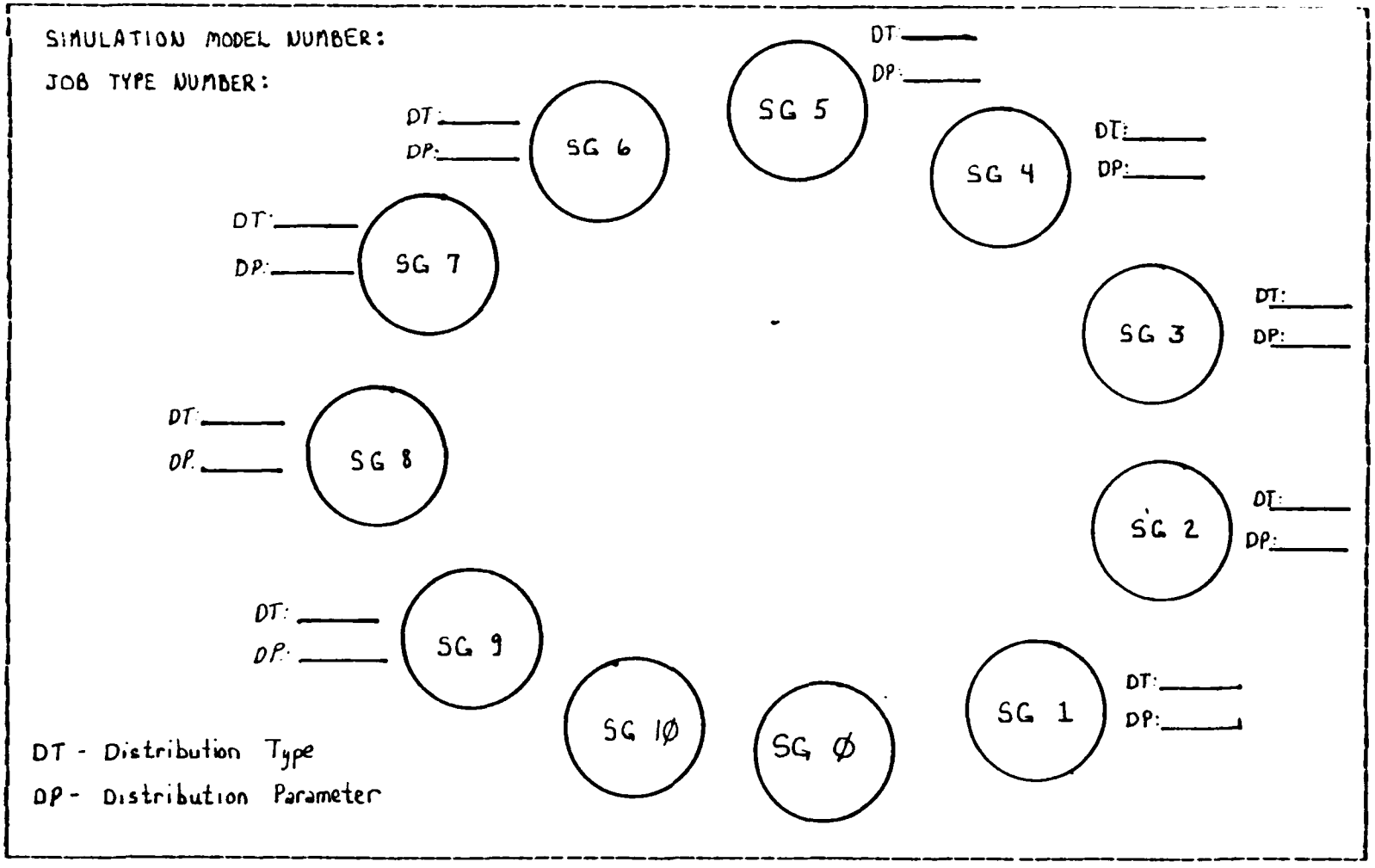


Figure 2.5 Job Type Routing Diagram Form.

2. Data Input Forms

Figure 2.6, the Server Group Record Data Form, and Figure 2.7, the Job Type and Routing Record Data Form are designed to group the model specification parameters in a format which facilitates the online input of data using CP¶T. The user should fill out one Server group Record Data Form per simulation model, and one Job Type and Routing Record Data Form for each job type in the model.

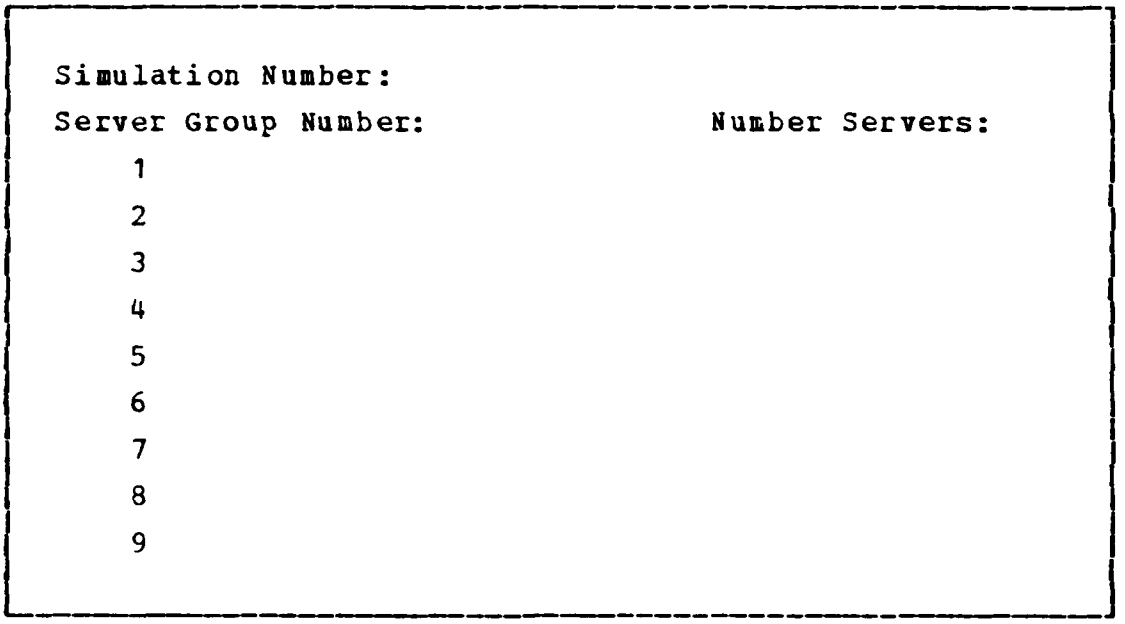


Figure 2.6 Server Group Data Form.

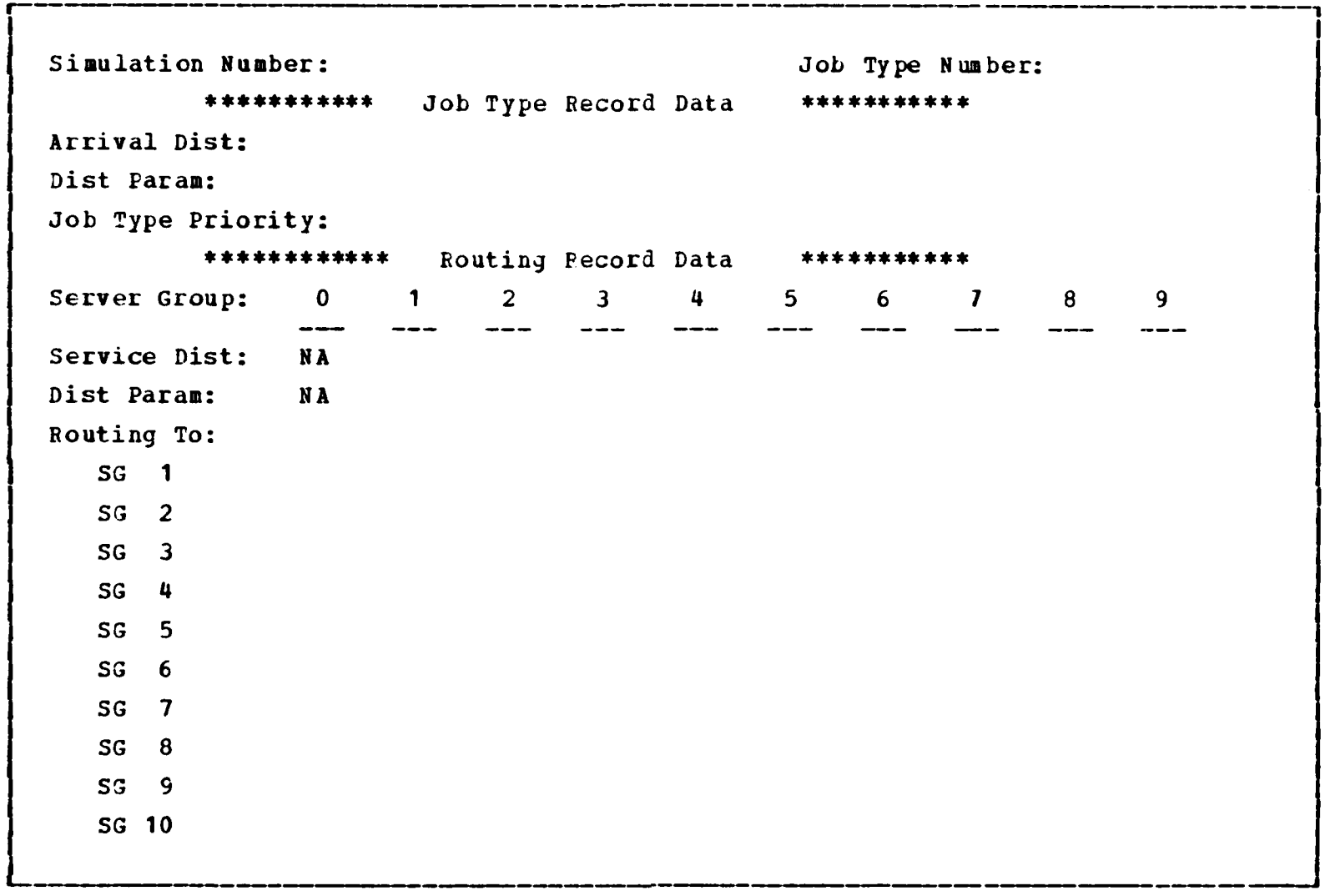


Figure 2.7 Job Type and Routing Record Data Form.

E. MODEL DESIGN EXAMPLE

In this section the model design process is illustrated through development of the required parameters to simulate a simple computer system consisting of a CPU and two disk drives. The computer system which forms the basis of the model design example is illustrated in Figure 2.10 and taken from [Ref. 1: pp. 168 - 174]. The analytic solution to the model is compared to the the CPMT model simulation results in Chapter 6.

1. Determine Data Input Parameters

Input parameters are developed for the three record types.

- **Simulation Model Number:** The simulation model is assigned the number '20' for identification purposes. The number was selected from numbers 1 through 99 not already in use to designate a simulation model in the CPMT data base.

Server Group Record:

- **Server Group Assignment:** The system being modeled consists of three server groups: the CPU, disk #1, and disk#2. For CPMT model input in this example, the following server group designations are made:

  CPU - Server Group #1

  Disk #1 - Server Group #2

  Disk #2 - Server Group #3

- **Number of Servers:** There is only one server in each of the three server groups.

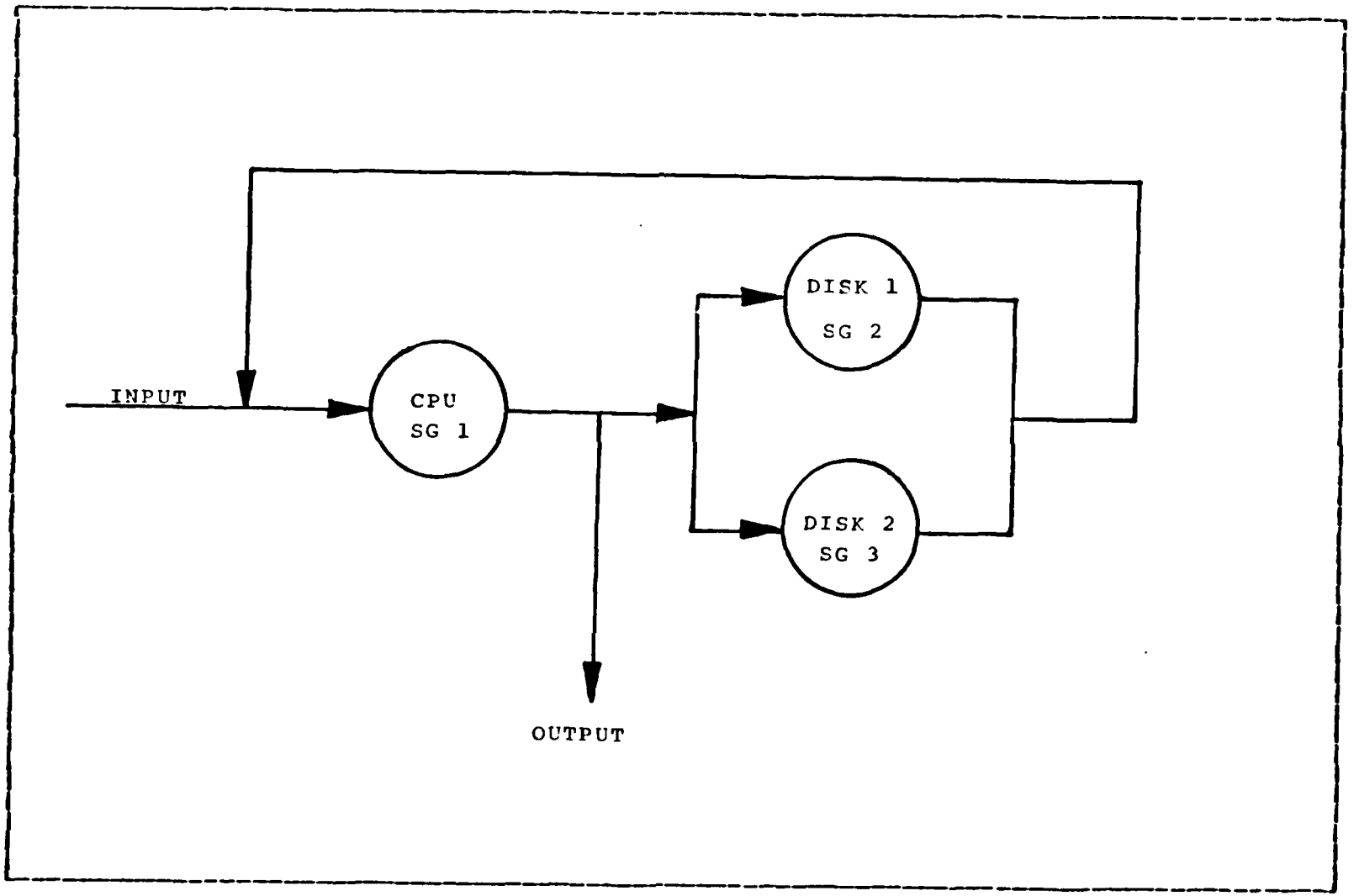


Figure 2.8 Diagram of Simulated Computer System.

Job Type Record:

- **Job Type Number:** The computer system has only one job type which will be designated as Job Type 1.
- **Arrival Distribution:** Assumed to be exponential in the analytic model.
- **Distribution Parameter:** The average job rate is given as 5 jobs per second. Given that the mean is 1/a, the mean = .2 seconds per job.
- **Job Type Priority:** Since there is only one job type in the system, the priority assigned to the job type is insignificant and will not affect the statistical results of the simulation. For the example, a priority of '1' is arbitrarily assigned to the job type.

Routing Records:

Four routing records are required to describe the routing of the job through the system and the service times at each of the server groups: one for each of the three server groups, and one for the "dummy" entrance server group: SG 0.

Server Group 0 Routing Record:

- **Service Rate:** Since SG 0 is the arrival server group, no processing is done at SG 0 and no values are assigned to the service distribution and distribution parameter.
- **Routing Probability:** The job always begins processing at the CPU, so the job will be routed with 100% probability to SG 1 from SG 0.

Server Group 1 Routing Record:

- **Service Distribution:** Exponential.

- **Distribution Parameter:** The mean service time of the job for each visit to the CPU is given as .009 seconds per job.
- **Routing Probability:** The average Job makes six visits to the CPU and is routed from the CPU six times: four times to Disk #2 (SG 3); once to Disk #1 (SG 2); and once to exit the system (SG 10). The probabilities are given in Figure 2.9. Routing probabilities from a given server group must be represented as integer values the sum of which equals 100. The calculated routing probabilities from Server Group 1 are rounded off to meet this input criterion.

SG #	Routing probability to SG		Prob Input Value
2	P(2) = x	P(2) = 16.67	17
3	P(3) = 4x	P(3) = 66.68	66
10	P(10) = x	P(10) = 16.67	17
	100 = 6x		100
	16.67 = x		

Figure 2.9 Routing Probabilities From SG 1.

Server Group 2 Routing Record:

- **Service Distribution: Exponential.**
- **Distribution Parameter:** The mean service time of the job for each visit to the CPU is given as .040 seconds per job.

- **Routing Probability:** The job will return to the CPU after processing, so the job will be routed with 100% probability to SG 1 from SG 3.

Server Group 3 Routing Record:

- **Service Distribution: Exponential.**
- **Distribution Parameter:** The mean service time of the job for each visit to the CPU is given as .025 seconds per job.
- **Routing Probability:** The job will return to the CPU after processing, so the job will be routed with 100% probability to SG 1 from SG 3.

2. Diagram and Check Model Parameters

The user may find it helpful to diagram the service rates and routing for each job type in the system by filling out the Job Type Routing Diagram Form provided in Figure 2.5. The Job Type Routing Diagram for the simulation model example is given in Figure 2.8. The diagram allows for an easy mapping of the required model data specifications to the job type record and routing record formats. Also, the diagram provides a visual display from which the user can verify that the model meets the routing design guidelines listed under the Routing Record subsection of Section A of this chapter.

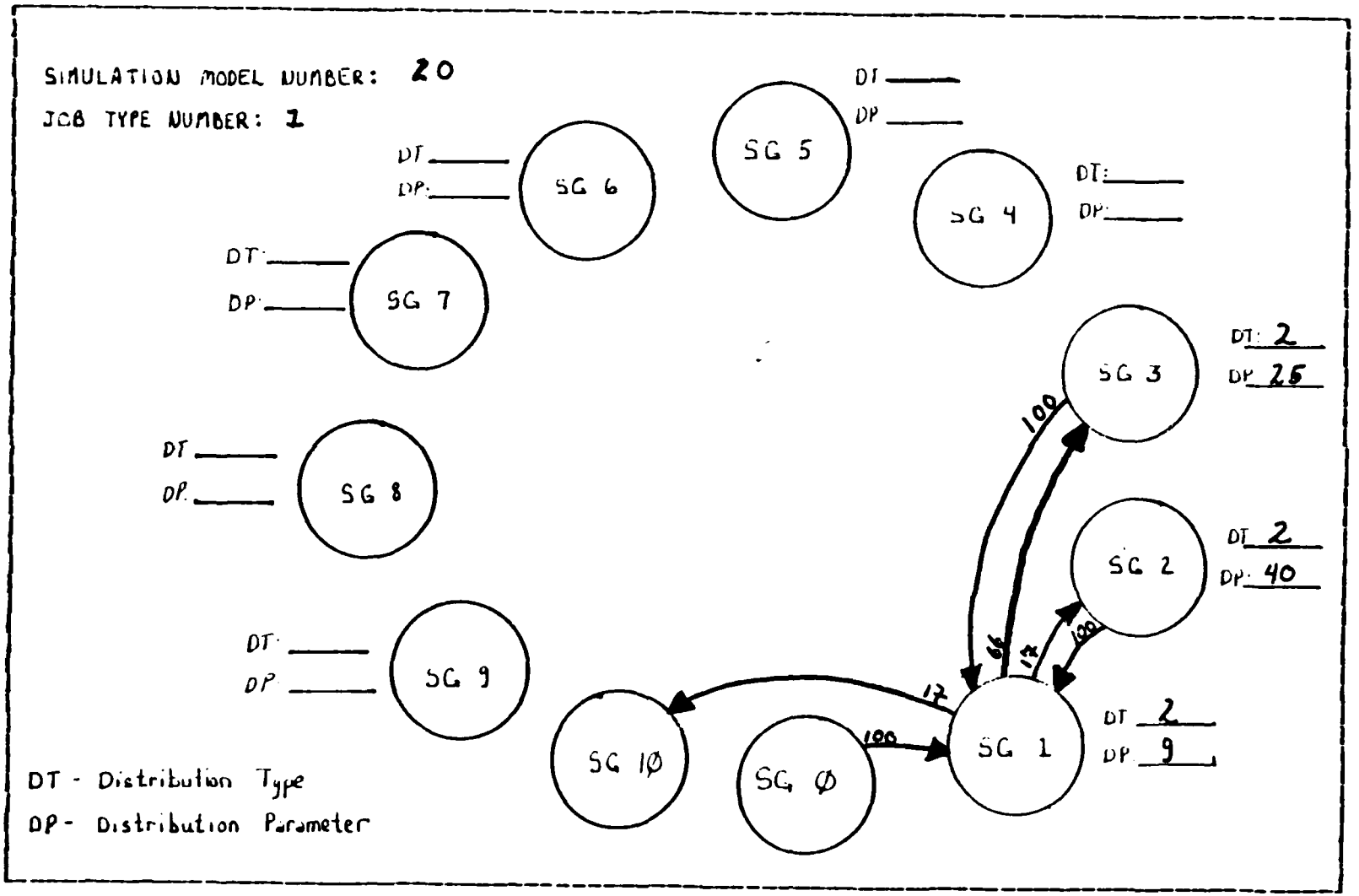


Figure 2.10 Job Type Routing Diagram.

3. Determine Time Unit

After the service and arrival rates have been determined, the model designer should look at all the values; determine what unit of time the "generic" time unit should equal; and then represent the values in the time unit chosen. In the example model, all times were originally calculated as fractions of seconds. The smallest amount of time represented in the model is the mean service time for the CPU: .009 seconds per job visit. Because the smallest time is in the millisecond range, the millisecond is selected as the time unit by which time values will be represented. All time values are multiplied by 1000 to result in a millisecond time representation.

4. Arrange Data in Record Format

To facilitate input of the model data parameters, the model data is entered into the input data record forms provided in Figures 2.6 and 2.7. The Server Group Record Data form for the model example is given in Figure 2.11 and the Job Type and Routing Record form is given in Figure 2.12.

Simulation Number : 20

Server Group Number:	Number Servers:
1	1
2	1
3	1
4	0
5	0
6	0
7	0
8	0
9	0

Figure 2.11 Server Group Record Data.

Simulation Number: 20 Job Type Number: 1

****** Job Type Record Data ******

Arrival Dist: 2

Dist Param: 200

Job Type Priority: 1

******** Routing Record Data ********

Server Group:	0	1	2	3	4	5	6	7	8	9
Service Dist:	NA	2	2	2						
Dist Param:	NA	9	40	25						
Routing To:										
SG 1	100	0	100	100						
SG 2	0	17	0	0						
SG 3	0	66	0	0						
SG 4	0	0	0	0						
SG 5	0	0	0	0						
SG 6	0	0	0	0						
SG 7	0	0	0	0						
SG 8	0	0	0	0						
SG 9	0	0	0	0						
SG 10	0	17	0	0						

Figure 2.12 Routing Record Form.

III. CPMT USER'S MANUAL

This section of the User's Manual is concerned with describing to the user the CPMT program itself: the options available to the user and how to use them. CPMT is an online interactive program controlled by dialogue. The information presented to the user in this section of the manual is meant to supplement and expand the instructions and options which are presented to the user online. In order to facilitate use, the User's Manual organization parallels the structure of the CPMT program.

A. RUNNING THE PROGRAM

CPMT runs on the VAX/VMS Computer Science Department Computer at NPS. To execute the program after logging onto the computer, enter the command

RUN CPMT

in response to the $ prompt. The program initially displays to the user the Master Menu of program options presented in Figure 3.1. The user enters the integer value corresponding to the option desired. A description of each of the options follows under separate headings.

A note of warning to the user: because of very strong typing in PASCAL, the CPMT program does not accept alphabetic character input when integer input is specified. If the user enters a non-integer character when an integer is expected, the program will abort. In this case, the user must restart the program.

At several points in the program, the user directs program control by responding to questions which have YES or NO answers such as

DO YOU WISH TO EXIT FUNCTION? Y/-

The convention for the CPMT program is that the user enters 'Y' for a YES response. Any other character input is interpreted as a NO response.

```
    COMPUTER PERFORMANCE MODELING TOOL (CPMT)
**********************************************************
Master Menu:

    1.   Update Data Base
    2.   Print Data Base
    3.   Check Simulation Specifications
    4.   Run Simulation Model
    8.   Exit CPMT
```

Figure 3.1 Master Menu.

B. UPDATE DATA BASE

It is under this option that the user enters the input data parameters for model specifications. This option updates an indexed sequential data base which can contain the specifications for many different simulation models. The data base is located in file RECFILE.DAT. The CPMT program accesses the file RECFILE.DAT in the directory in which the program is executing. If RECFILE.DAT does not exist in that directory, the program will automatically create the file. If the user wishes to initialize the data base, it is sufficient to delete the existing RECFILE.DAT file from their directory and have the program create a new

file during the next program execution. If the CPMT is copied to a new directory, the user of that directory can either copy an existing RECFILE.DAT file into the directory or have the program create a new file.

Each simulation model is identified by a unique integer value called a simulation number. The user may assign a simulation model an integer number between 1 and 99. Upon entering the update option, the program displays the prompt:

ENTER SIMULATION MODEL NUMBER OF MODEL YOU WISH TO UPDATE. VALID MODEL NUMBERS 1 THROUGH 99

At this point the user enters the simulation number for a new or an existing simulation model. All options for adding and deleting records from the data base are executed for the simulation number specified until the user changes the simulation number.

The update options are presented to the user in a menu format similar to that of the master menu. The update menu options are listed in figure 3.2.

1. Change Simulation Number

This function is used to change the simulation number. The user receives the same prompt as is displayed on first entering the Update option and responds by entering the simulation number desired.

2. Add Job Type Record

- The different job types for a given simulation number are numbered sequentially from 1 to 99. When the user specifies the 'Add Job Type' function, the program automatically accesses the simulation model data base to determine the next available job type number for the given simulation model and assign that number to the Job Type record to be added. The assigned job type

```
        UPDATE MODULE MAIN MENU
   1.   Enter New Simulation Model Number
   2.   Add Job Type Record
   3.   Add Routing Record
   4.   Add Server Group Record
   5.   Delete Job Type Record
   6.   Delete Routing Record
   7.   Delete Server Group Record
   8.   Copy Simulation Model
   9.   Delete Simulation Model
  10.   Exit Update Module
```

Figure 3.2 Update Menu Options.

number is displayed on the screen. Because of automatic job type numbering, the user should enter the job type records in the order corresponding to the job type numbering desired.

- The program then asks a series of questions requesting the user to input the ARRIVAL DISTRIBUTION, appropriate ARRIVAL DISTRIBUTION PARAMETER, and PRIORITY of job type. The program dialogue is presented in Figure 3.3. The data input parameters requested are those under the Job Type Record Data heading of the Job Type and Routing Record Data Form, Figure 2.7.

- Valid user input for the data fields requested is presented in figure 2.2 and presented to the user interactively. The program edits the input data values for validity and prompts for reentry of data which does not meet the edit check.

```
          UPDATE MODULE
          ADD JOB TYPE RECORD FUNCTION
**********************************************************

SIMULATION MODEL NUMBER:
JOB TYPE NUMBER:

ENTER THE JOB TYPE ARRIVAL DIST:      1. DETERMINISTIC
                                      2. EXPONENTIAL
                                      3. UNIFORM

ENTER DETERMINISTIC VALUE: *
ENTER EXPONENTIAL DISTRIBUTION MEAN: *
ENTER UPPER BOUND OF UNIFORM DISTRIBUTION: *

ENTER PRIORITY FOR THIS JOB TYPE:
VALID PRIORITY CODES ARE 1 THROUGH 10.

* Distribution parameter request
  depends on distribution type
```

Figure 3.3 Add Job Type Record Dialogue.

- After the job type data is entered, the program displays the record data for user review. At this point the user has the option of adding the record to the data base. If the user response indicates a desire to add the record, then the program should respond with the message **RECORD SUCCESSFULLY ADDED.** If the user does not choose to add the record the program displays the message **RECORD NOT ADDED.**
- If the record is successfully added to the data base, the user has the option of going immediately into the Add Routing Record function (see next section) to add the routing records which are associated with the job

type record just added. If the user enters the Add Routing Record Function from the Add Job Type Record function, program control returns to the Add Job Type function on exit from the Add Routing Record function.

- The user may enter multiple job type records for a given simulation number while in the Add Job Type Function. At the end of every iteration of the function dialogue the user is given the option of exiting the function. Upon exit, control is returned to the Update Menu.

3. Add Routing Record

- This function can be entered either from the Add Job Type record function or from the Update Menu directly. When the function is exited, the user is returned to the part of the program from which the function was entered.
- If the user entered the function from the Add Job Type record function, then the function will automatically add the routing records for the job type number of the record just added.
- If the user entered the function from the Update Menu, then the program will ask the user to identify the job type number for which the routing records are being added. If the job type record for the identified job type number does not exist on the data base, then the program will not allow routing records to be added. In this case a message will be sent to the terminal indicating that the job type record for the specified job type number does not exist.
- The program prompts the user for input of the following data items: the SERVER GROUP NUMBER, the SERVICE DISTRIBUTION, appropriate SERVICE DISTRIBUTION PARAMETER, and the ROUTING PROBABILITY which indicates

the routing probability from the server group which is the subject of the routing record to the other server groups in the system. The function dialogue is presented in Figure 3.4. The data parameters requested correspond to those under the Routing Record Data heading of the Job Type and Routing Record Data Form of Figure 2.7.

```
        UPDATE MODULE
        ADD ROUTING RECORD FUNCTION
**********************************************

ENTER JOB TYPE NUMBER OF ROUTING RECS TO BE ADDED:*

ENTER ROUTING RECORD SERVER GROUP NUMBER:

ENTER SERVICE RATE DISTRIBUTION:   1. DETERMINISTIC
                                   2. EXPONENTIAL
                                   3. UNIFORM

ENTER DETERMINISTIC VALUE: **

ENTER EXPONENTIAL DISTRIBUTION MEAN: **

ENTER UPPER BOUND OF UNIFORM DISTRIBUTION: **

ENTER THE ROUTING PROBABILITY FROM SERVER GROUP __ TO
SERVER GROUP  1:
SERVER GROUP  2:
SERVER GROUP  3:
SERVER GROUP  4:
SERVER GROUP  5:
SERVER GROUP  6:
SERVER GROUP  7:
SERVER GROUP  8:
SERVER GROUP  9:
SERVER GROUP 10:

* Asked if entered from main menu

** Distribution parameter request depends on
   distribution type.
```

Figure 3.4 Add Routing Record Dialogue.

- Valid user input for the data fields requested is presented in Figure 2.3. The program edits the SERVICE DISTRIBUTION and SERVICE DISTRIBUTION parameter input data values for validity and prompts for reentry reinput of data which does not meet the edit check. Also, the program checks to ensure that the sum of the routing probabilities equals 100. If they do not the message **ROUTING PROBABILITIES DO NOT ADD UP TO 100 PLEASE REENTER ALL PROBABILITIES** is displayed and the user must reenter probabilities.

- After the routing record data is entered, the program displays the record data for user review. The user has the option of adding the record to the data base. If the user desires to add the record, the program will attempt to add the record. If the record is successfully added, the message **RECORD SUCCESSFULLY ADDED** is displayed. If the add attempt fails because of the existence of another record with the same record key, the message **RECORD ALREADY EXISTS, NOT ADDED** is displayed. If the user chooses not to add the record, the message **RECORD NOT ADDED** is displayed.

- The Add Routing Record function loops so that the user may input multiple routing records for a given simulation number job type before exiting the function.

4. Add Server Group Record

- The dialogue requests that the user input the NUMBER OF SERVERS for each server group in the system. The dialogue is presented in Figure 2.11. The data input parameters requested correspond to the Server Group Data Form, Figure 2.6.

```
        UPDATE MODULE
        ADD SERVER GROUP RECORD FUNCTION
**************************************************

SIMULATION MODEL NUMBER:

ENTER NUMBER OF SERVERS FOR
SERVER GROUP 1:
SERVER GROUP 2:
SERVER GROUP 3:
SERVER GROUP 4:
SERVER GROUP 5:
SERVER GROUP 6:
SERVER GROUP 7:
SERVER GROUP 8:
SERVER GROUP 9:
```

Figure 3.5 Add Server Record Dialogue.

- After the server group record data is entered, the program displays the record data for user review. The user has the option of adding the record to the data base. If the user desires to add the record, the program will attempt to add the record. If the record is successfully added, the message **RECORD SUCCESSFULLY ADDED** is displayed. If the add attempt fails because of the existence of another record with the same record key, the message **RECORD ALREADY EXISTS, NOT ADDED** is displayed. If the user chooses not to add the record, the message **RECORD NOT ADDED** is displayed.

5. Delete Job Type Record

- Dialogue requests the Job Type number of the job type record to be deleted.
- If the job type record is in the data base then the record is displayed on the screen and the user is given the option of deleting it. If the user chooses to delete it, the message **RECORD DELETED** is displayed. If

the user does not delete it, the message **RECORD NOT DELETED** is displayed.

- If the record does not exist in the data base, the message **NO RECORD FOUND** is displayed.
- When a job type record is deleted, all the routing records which are subordinate to that job type are also deleted.
- The user has the option of exiting the function at the end of every iteration of the function dialogue.

6. Delete Routing Record

- Dialogue asks user to input the Job Type number and the server group number for the record to be deleted.
- If the job type record is in the data base then the record is displayed on the screen and the user is given the option of deleting it. If the user chooses to delete it, the message RECORD DELETED is displayed. If the user does not delete it, the message RECORD NOT DELETED is displayed.
- If the record does not exist in the data base, the message RECORD NOT FOUND is displayed.

7. Delete Server Group Record

- Since there is only one server group record per simulation model, the user is not required to input any further parameters.
- If the server record is in the data base then the record is displayed on the screen and the user is given the option of deleting it. If the user chooses to delete it, the message **RECORD DELETED** is displayed. If the user does not delete it, the message **RECORD NOT DELETED** is displayed.

- If the record does not exist in the data base, the message **NO RECORD FOUND** is displayed.

- Since there is only one server record per simulation model, the function provides no looping mechanism. The user returns to Update Menu when ready by entering a character.

8. Copy Simulation Model

- The copy function copies all the records for a given simulation number to a new simulation number. The copy option is convenient if the user wishes to change a few parameters of a model design and compare the simulation results of the original and modified models. In this case, the user can copy the original model to a new model number, make the parameter changes in the copy, and maintain both model design specifications in the data base.

- The copy function is not subject to a previously entered simulation model number, as are the record addition and deletion options. The user enters the simulation model number of the model which is being copied and the model number of the new copy. If the copy is successful, the message **SIMULATION MODEL COPIED** is displayed. If the number of the model being copied does not exist, the message **SIMULATION MODEL NUMBER __ DOES NOT EXIST ON DATA BASE** is displayed. If the new simulation model number is already on the data base, the message **SIMULATION MODEL NUMBER __ ALREADY EXISTS** is displayed and the model is not copied.

9. Delete Simulation Model

- The delete simulation model function deletes all the records for a given simulation model number from the data base.

- The delete simulation model function is not subject to the previously entered simulation model number. The user enters the number of the model to be deleted in response to the program prompt.
- After the user enters the model number, the program attempts to find the number in the data base. If the simulation model number does not exist, the message **SIMULATION MODEL NUMBER __ DOES NOT EXIST** is displayed. If the simulation model number is found, the program gives the user the option of deleting the model.

10. Exit

Upon selection of this option, control is returned to the Master Menu from the Update Menu.

C. PRINT DATA BASE

Upon selection of this option, a printout of the entire indexed sequential data base is written to file OUTFILE.DAT. If the user desires a hard copy, the user can request a print of the file from outside the CPMT environment.

D. CHECK SIMULATION SPECIFICATIONS

For this option the user supplies the simulation model number of the model to be checked in response to the system prompt: **ENTER NUMBER OF SIMULATION MODEL TO CHECK.** The program then executes the procedure which checks the simulation model records to ensure that certain requirements are met with respect to existence of necessary records and routing relationships. The simulation model must have a header record, server group record, and a minimum of one job type record and associated routing records. Additionally, the routing records must meet the guidelines outlined in the

Routing Record subsection of Chapter 2. A simulation model will not execute unless it passes the simulation specification check.

If the simulation model meets the requirements the message SIMULATION MODEL SPECIFICATIONS CHECK is displayed. If the model does not meet the requirements then appropriate error messages are written to the file OUTFILE.DAT to indicate to the user the model specification deficiencies. The list of possible error messages is presented in Figure 3.6.

```
1.  Simulation Number Does Not Exist.
2.  No Server Group Record Exists.
3.  No Job Type Record Exists.
4.  Job Numbers Are Not Sequential.
5.  Server Group __, Job Type __ Routing Loop.
6.  No Routing Records Exist for Job Type __.
7.  No Server Group 0 Routing Record For Job Type __.
8.  Job Type __ not Routed to Exit Server Group.
9.  Job Type __ Routed To But Not From Server Group __.
10. No Server Group 0 Routing Record For Job Type __.
```

Figure 3.6 Simulation Specification Error Messages.

E. EXECUTE SIMULATION MODEL

The program will prompt the user to input the number of the simulation model to be run, the number of jobs to be run for the simulation, and a seed value. The program dialogue is presented in Figure 3.7.

```
ENTER SIMULATION NUMBER OF MODEL TO EXECUTE
ENTER NUMBER OF JOBS TO RUN IN SIMULATION
ENTER SEED YOU WANT TO USE
```

Figure 3.7 Execute Simulation Model Dialogue.

The seed will be used as initial input into the random number generator. The random numbers in turn are used as input into functions which generate random variates according to user specified distributions.

The program will check the simulation specifications and execute the simulation model if the specifications check. If the specifications do not check, error messages are written to file OUTFILE.DAT. See the previous section for further details. If the execution is successful, the message **SIMULATION MODEL EXECUTED. OUTPUT STATISTICS IN FILE OUTFILE.DAT is displayed.** The statistical output will be written to file OUTFILE.DAT. An example of a simulation run output report is provided in Figure 3.8. A description of the output report statistics and their origination is presented in the Execute and Tabulate Module section of Chapter 4. The user may obtain a hard copy print of the file by requesting a print outside the CPMT environment.

F. EXIT

Exits the CPMT environment. The program execution is terminated and control returns to the system.

SIMULATION NUMBER IS 25

SEED IS 89274

NUMBER JOBS RUN IS 1000

JOB TYPE	MAX QTIME	MIN QTIME	MEAN QTIME	STDD QTIME	MAX STIME	MIN STIME	MEAN STIME	STDD STIME
ALL	2442	0	129.409	260.888	4108	1	323.411	428.061
1	2442	0	129.409	260.888	4108	1	323.411	428.061

SERVER GROUP	MAX QLEN	MIN QLEN	AVG QLEN	SG UTIL	SERVER NUMB	SERVER UTIL
1	6	0	0.105	0.268		
					1	0.268
2	4	0	0.053	0.193		
					1	0.193
3	9	0	0.463	0.471		
					1	0.471

Figure 3.8 Simulation Run Statistical Report Example.

IV. PROGRAM SPECIFICATIONS

The purpose of this chapter is to provide a general description of the main modules and procedures which comprise the CPMT program, with emphasis on an overview of program processing, dynamic record structures used during program execution, and data structure interfaces between the main program modules. The first five sections of this chapter discuss the CPMT main driver and the four main program modules. The general description presented in these sections is designed to complement the more detailed inline comments in the PASCAL source code. The next section presents a data dictionary of the dynamic records used by the CPMT program. The final section of the chapter lists the physical PASCAL source code and data files which comprise the CPMT.

A. CPMT MAIN DRIVER

The main driver program controls the master program loop which displays the Master Menu to the user and processes user options. The program driver uses a case statement structure to branch to appropriate procedures. The procedures corresponding to the Master Menu options are listed below.

1. Update Data Base option. The driver calls procedure UPDATE_MENU, which is the main control procedure for the Update Module.
2. Print Data Base option. The driver calls the PRINT_DATA_BASE procedure. The PRINT_DATA_BASE procedure reads sequentially through the entire data base file RECFILE.DAT and formats the records into an output report written to file OUTFILE.DAT.

3. Check Simulation Specifications o tion. The driver calls the CHECK_SIM_SPECS procedure.
4. Run Simulation Model option. The driver first calls the CHECK_SIM_SPECS procedure. If the model to be executed passes the check procedure then the driver calls the CREATE_JOBSTREAM procedure followed by the EXECUTE_AND_TABULATE procedure to execute the simulation model.
5. Exit option. The driver exits the main control loop and terminates program execution.

The main modules and procedures of the CPMT are discussed in the following sections in further detail.

B. UPDATE MODULE

1. General Description

The Update Module controls the interactive input of data from the terminal to create an indexed sequential data base of simulation model parameters.

2. Input

Input into the Update Module consists of data parameters and program control commands which are entered by the user on the terminal. Chapter 3, the User's Manual, gives a detailed description of the input options.

3. Output

Output of the module is an updated indexed sequential data base consisting of job type, routing, and server group records which contain the data required to run a simulation model. A detailed organization of the indexed sequential data base is given in Chapter 5, Data Base Specifications.

4. Files Accessed

The Update Module accesses two files:

- RECFILE is the file which contains the indexed sequential data base.
- MESSAGES is a sequential file which contains dialogue messages sent to the terminal to communicate with the user.

5. Processing

The Update Module driver executes a loop which presents the Update Menu to the user and processes user options. The program processes selected options using a case command which calls the appropriate procedure to handle the requested option. Update Module user options currently include the addition and deletion of server group, job type and routing records from the data base; copying an entire simulation model to a new simulation model number; deleting an entire simulation model from the data base; and changing the simulation model number for which record addition and deletion requests are processed. The User's Manual of Chapter 3 presents a detailed description of the user options available in the Update Module and the dialogue and input parameters for each option. For each of the update options, the Update Module procedures control the interactive dialogue requesting the necessary data items. The procedure then updates the indexed sequential data base with the data supplied by the user. Program access of the data base is performed using the PASCAL file processing commands discussed in Chapter 5.

C. CHECK SIMULATION SPECIFICATIONS MODULE

1. General Description

This procedure checks the record parameters of a simulation model number for the conditions necessary to ensure that the model can be executed by the Execute and Tabulate Module.

2. Input

Input into the procedure is the indexed sequential data base file RECFILE.DAT and an integer parameter passed by value which indicates the number of the simulation model to be checked.

3. Output

Output from the procedure is the boolean variable CHECK, passed by reference, which is set to true if the simulation model passes the check procedure and set to false if it fails the check. If the simulation model does not check, error messages are written to the file OUTFILE.DAT. Possible error messages are listed in Figure 3.6.

4. Files Accessed

The Check Simulation Specifications Module accesses two files:

- RECFILE is the file which contains the indexed sequential data base.
- OUTFILE is a sequential file to which error message output is written.

5. Processing

The procedure accesses the data base to find the header record of the simulation model being checked. It then sequentially reads the data base until it has read all the records with the simulation model number being checked. As the simulation model records are processed, the procedure checks to ensure that a header record, server group record, and at least one job type record exist for the model number. It further checks to ensure that the job type records are sequentially numbered, and that the routing specifications for each job type meet the routing design requirements outlined under the Routing Record subsection of Chapter 2.

D. CREATE JOB STREAM MODULE

1. General Description

The main purpose of this module is to generate the jobs which will be used as input to the simulation execution. The module uploads the simulation model job type and routing records from the data base into a dynamic linked list of job type and routing records. The job type/routing linked list is illustrated in Figure 4.1. The module then accesses the job type and routing parameters of the linked list and generates jobs from the parameters.

2. Input

Input into the Create Job Stream Module is the indexed sequential data base RECFILE.DAT.

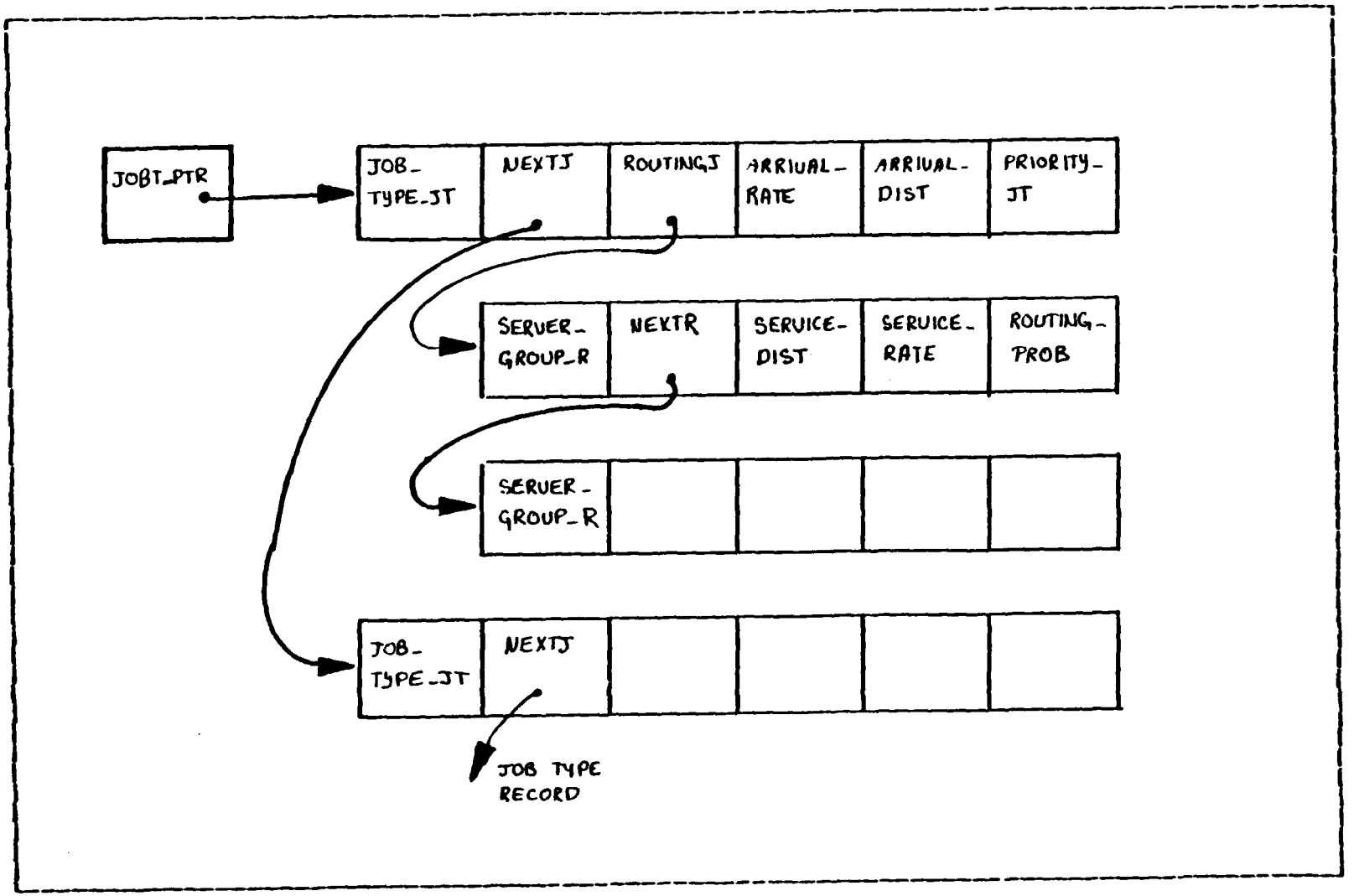


Figure 4.1 Job Type/Routing Linked List.

3. Output

Output of the module is a dynamic linked list of job and event records which will serve as input to the Execute and Tabulate Module. Figure 4.2 is a diagram of the job/event linked list record structure.

4. Files Accessed

The Create Job Stream Module accesses the indexed sequential data base in file RECFILE.DAT.

5. Processing

The Create Job Stream Module (CJS) initially executes the BUILD_LL_FROM_DB procedure. This procedure uploads the job type and routing simulation model records from the data base file into a dynamic linked list record structure of job and routing records illustrated in Figure 4.1. The fields of the job type and routing records are explained in detail in the Data Dictionary section of this chapter. The procedure also reads the server group record data into the a variable array of integers named NUM_SERVERS. The job type/routing linked list is used in the Create Job Stream Module to create jobs which will be run by the system. The NUM_SERVERS array is used by the Execute and Tabulate Module CREATE_SERVER_GROUP procedure to provide data on the server groups and servers which must be created to run the simulation.

The program then enters a loop to build the linked list of job and event records which will become the arrival queue for the Execute and Tabulate Module. The job/event linked list record structure is diagramed in Figure 4.2. The fields of the job and event records are explained in detail in the Data Dictionary section of this chapter. The job/event arrival queue is pointed to by the variable

ARRIVEQPTR. The program adds jobs to the arrival queue in ascending order by their arrival times until the number of jobs in the queue is equal to the total number of jobs being run through the simulation execution.

E. EXECUTE AND TABULATE MODULE

1. General Description

The Execute and Tabulate Module (EXT) processing can be divided into three major parts: the creation of the linked list of server group and server records; the processing of the jobs in the arrival queue through the server group and server structure and the concurrent statistical data gathering; and the calculation and preparation of the statistical output report of the simulation results.

2. Input

Input into the EXT module is the arrival queue linked list of job and event records produced by the CJS module, and the NUM_SERVERS array which contains the number of servers in each server group for the simulation model being run. The NUM_SERVERS array is loaded by the BUILD_LL_FROM_DB procedure in the Create Job Stream Module.

3. Output

Output from the module is a formatted statistical report of the simulation run which is written to file OUTFILE.DAT.

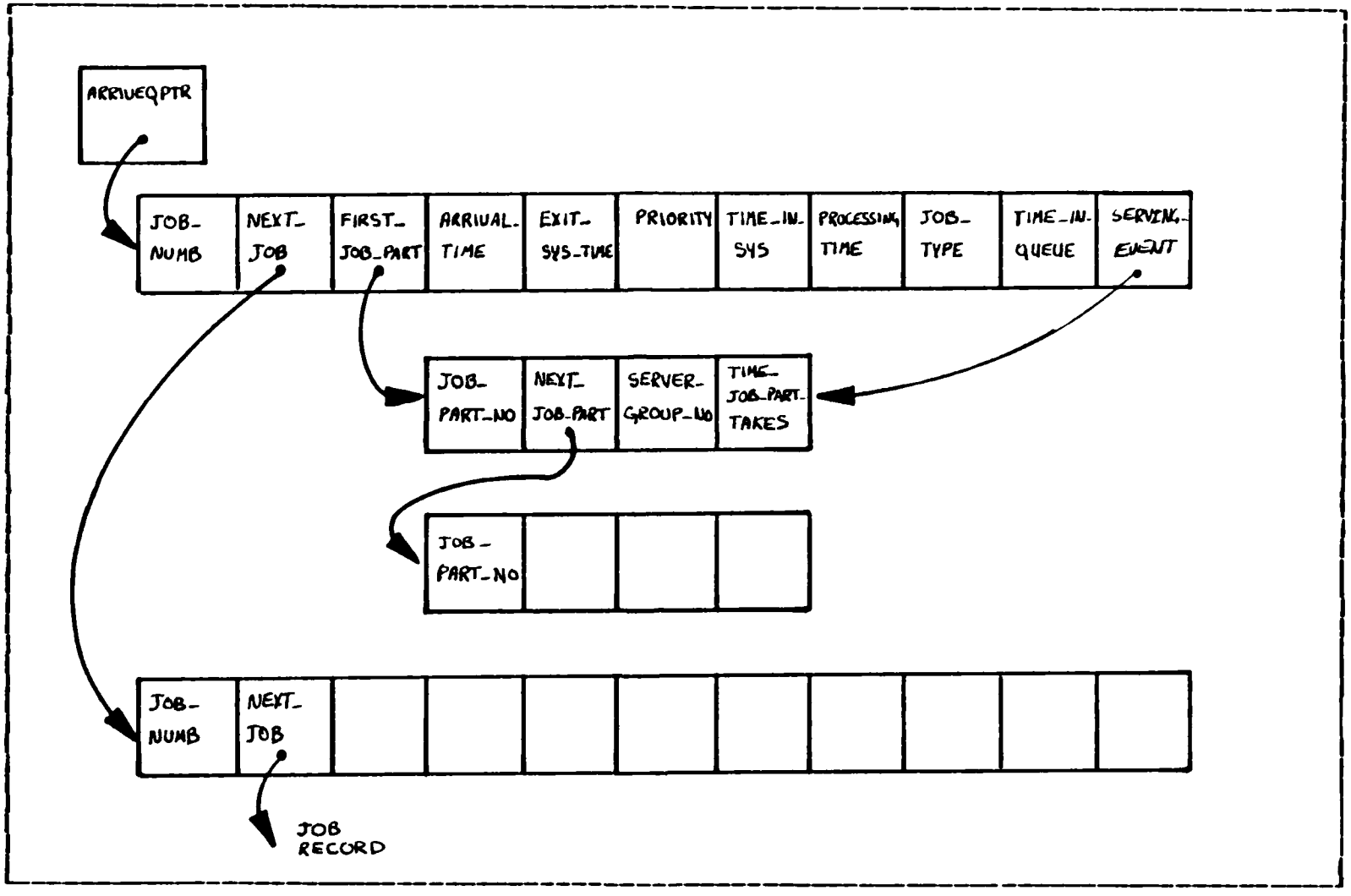


Figure 4.2 Job/Event Linked List.

4. Files Accessed

The Execute and Tabulate Module writes the simulation run output report to file OUTFILE.DAT.

5. Processing

The EXT Module calls procedure CREATE_SERVER_GROUPS to create the linked list record structure of server group and server records which will be used to simulate the modeled computer. The server group/server linked list record structure is diagramed in Figure 4.3. The fields of the server group and server records are explained in detail in the Data Dictionary section of this chapter. A server group record is always created for Server Group 0, the 'dummy' arrival server group. The arrival queue of job records is attached to the FIRST_IN_Q pointer field of the Server Group 0 record. No server records are created for Server Group 0 because no event processing occurs there. The procedure next creates server group and server records for each of the utilized server groups in the system. The CREATE_SERVER_GROUP procedure accesses the data on the number of server groups and servers in the system from the NUM_SERVERS array. The server group records are linked together by the NEXT_SERVER_GROUP pointer in ascending order by server group number (server group record field SERVER_GROUP). A server record is created for each server in the server group. The FIRST_SERVER pointer field of the server group record points to the first server record in the server group. Subsequent server records are linked via the NEXT_SERVER pointer field in the server records. The pointer variable FIRST_SGPTR points to the first server group record in the server group/server linked list, which is always Server Group 0.

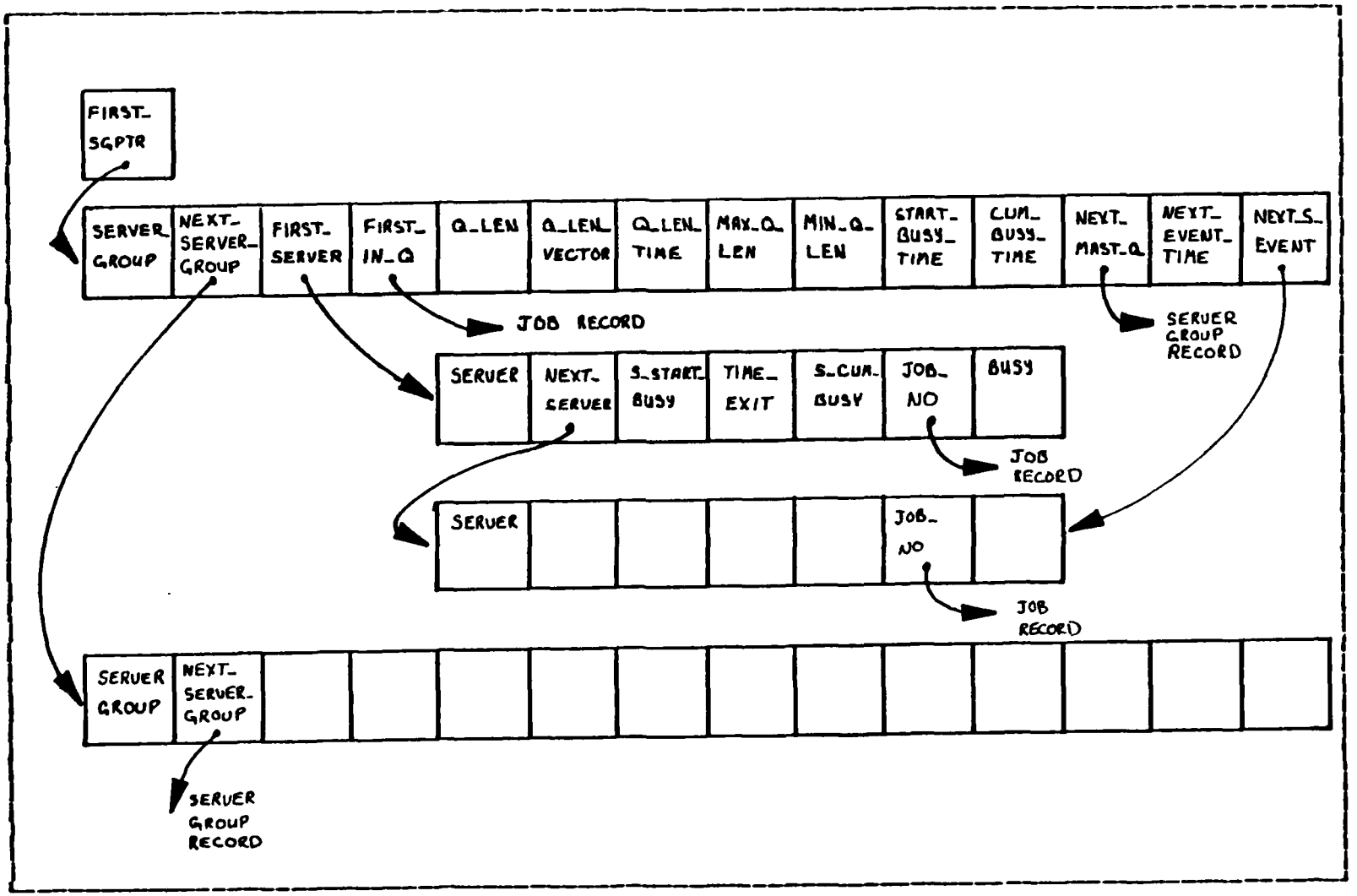


Figure 4.3 Server Group/Server Linked List.

After the server group linked list record structure is in place, the program begins execution of the main loop which processes the jobs through the server groups. The timing sequence of the job processing is monitored by a clock integer variable named CLK and ordered by the Master Event Queue linked list. The Master Event Queue linked list is pointed to by the pointer variable MASTERQPTR. It is an ordering of the busy server group records in ascending order by time of the next event which occurs at the server group. The server group next event time is indicated in the NEXT_EVENT_TIME field of the server record. The server records are linked in the Master Event Queue by the NEXT_MAST_Q pointer field of the server group record.

At the beginning of the job processing, the CLK variable is set to '0' and the MASTERQPTR points to Server Group 0, since the first event will be the arrival of the first job into the system. The process of moving jobs through the system is handled by a series of job departures and job arrivals at the system server groups. The loop begins with a job departure from the Server Group which is first in the Master Event Queue because it has the next job event time. The departure job is detached from the server group and the server group is detached from the Master Event Queue. The server group is then updated and reinserted into the Master Event Queue in order by its revised next event time. If the server group becomes idle, it is not reinserted into the master event queue.

The job which was just detached as a departure then becomes an arrival. The procedure locates the server group at which the job is arriving by traversing the Server Group/Server linked list record structure pointed to by the FIRST_SGPTR until it finds the target server group. If the server group is in the Master Event Queue it is detached from the queue because of the possibility that its

Next_Event_Time will change with the new job arrival. The arrival job is then attached to the server group, the server group and server records are updated, and the server group is then inserted into the Master Event Queue by its NEXT_EVENT_TIME. The procedure continues to process the jobs through the Server group structure as a series of job departures and job arrivals until the MASTERQPTR is nil, which indicates that all events have been processed.

After all jobs and events are processed through the Master Event Queue, the EXT module calculates and formats the statistical output report. An example of the simulation run output is provided in Figure 3.8. The statistics generated for the output report fall into two categories: job type statistics and server group/server statistics. A description of the report statistic items and their origination follows.

Job Type Output Statistics:

The job type statistics are calculated for all the jobs in each job type category and for all the jobs in the system. The job type statistics presented in the output report are based on two fields in the dynamic job records of the completed jobs: the TIME_IN_QUEUE field and the TIME_IN_SYS field. The TIME_IN_QUEUE field represents the total time a given job spent in all server group queues between the time it entered and exited the system. The TIME_IN_SYS field represents the total time the job spent in the system, or the difference between the job's exit and arrival times.

The report statistics based on the TIME_IN_QUEUE fields of the job records include MAX QTIME, MIN QTIME, MEAN QTIME, and STDD QTIME. MAX QTIME is the maximum time any job of the given category spent waiting in server group queues. It represents the greatest value of the

TIME_IN_QUEUE fields of all the job records for the indicated category. MIN QTIME is the minimum time a job spent in server group queues and represents the smallest value of the TIME_IN_QUEUE fields of all the job records for the indicated category. MEAN QTIME is the average of all the TIME_IN_QUEUE fields for job records in the indicated category. STDD QTIME is the standard deviation of the TIME_IN_QUEUE fields for jobs of the indicated category.

Job type output report statistics based on the TIME_IN_SYS fields of the completed job records include: MAX STIME, MIN STIME, MEAN STIME, and STDD STIME. MAX STIME is the maximum time a job of the indicated category spent in the system. It represents the greatest value of all the TIME_IN_SYS fields of the job records in the category considered. MIN STIME is the minimum time that a job spent in the system and it represents the smallest value in the TIME_IN_SYS fields of the job records considered. MEAN STIME is the average of the TIME_IN_SYS fields for all jobs in the category, and STDD STIME is the standard deviation of the TIME_IN_SYS fields.

Server Group/Server Statistics:

The server group and server statistics are presented for each server group in the simulation model. The server group statistics include: MAX QLEN, MIN QLEN, AVG QLEN, and SG UTIL. For each server in the server group the SERVER UTIL calculation is also presented. MAX QLEN represents the maximum queue length for the given server group during the simulation run execution. MAX QLEN is taken from the MAX_Q_LENGTH field of the dynamic server group record. MIN QLEN represents the minimum queue length for the server group and is taken from the MIN_Q_LENGTH field of the server group record. AVG QLEN represents the mean queue length during the simulation execution. AVQ QLEN is calculated by

dividing the Q_LEN_VECTOR field of the server group record by the total system run time. Q_LEN_VECTOR field is explained in the following section. SG UTIL is the Server Group utilization. It is calculated by dividing the server group cumulative busy time (server group record field CUM_BUSY_TIME) by the total system run time. SERVER UTIL is the server utilization and it is calculated by diving the server cumulative busy time (server record field S_CUM_BUSY) by the total system run time.

F. DATA DICTIONARY OF DYNAMIC RECORD ITEMS

The Create Job Stream and Execute and Tabulate modules use six dynamic record types: the job type, routing, job, event, server group and server records. The records form three dynamic record structures: the job type/routing linked list (Figure 4.1), the job/event linked list (Figure 4.2) and the server group/server linked list (Figure 4.3). The data fields in the dynamic records are described in the following sections.

1. Job Type Record

JOB_TYPE_JT. Integer. The job type field value corresponds to the Job Type Number field of the data base Job Type record. The Job Type Number is an integer value from 1 to 99 assigned to each job type for purposes of identification. The BUILD_LL_FROM_DB procedure sequentially assigns the JOB_TYPE_JT numbers commencing with '1' as the dynamic job type records are uploaded from the data base Job Type records.

NEXTJ. Pointer. The Next Job Pointer points to the next job record. The NEXTJ pointer links the Job records in ascending order by JOB_TYPE_JT job field.

ROUTINGJ. Pointer. The ROUTINGJ pointer points to the first routing record subordinate to the job type record.

ARRIVAL_RATE. Integer. The BUILD_LL_FROM_DB procedure reads the value of the data base Job Type Arrival Distribution Parameter field into the dynamic job type record ARRIVAL_RATE field. See Figure 2.4 for a description of the distribution parameter field.

ARRIVAL_DIST. Integer. The BUILD_LL_FROM_DB procedure reads the value of the data base Job Type Arrival Distribution field into the dynamic job type record ARRIVAL_DIST field. Figure 2.4 lists the three distribution types and their integer codes.

PRIORITY_JT. Integer. The BUILD_LL_FROM_DB procedure reads the value of the data base Job Type Priority field into the dynamic job type PRIORITY_JT field. Valid Job Type Priority range is from 1 to 10, with 1 being the highest.

2. Routing Record

SERVER_GROUP_R. Integer. The BUILD_LL_FROM_DB procedure assigns to the SERVER_GROUP_R field the value of the Server Group Number of the data base Routing record, which is actually the RR_S_NUM record key component of the data base Routing record. See Chapter 5 for an explanation of the data base record key components. The Server Group Number of the data base Routing record identifies the server group to which the routing record data pertains. Valid Server Group Numbers range from 0 to 9.

NEXTR. Pointer. The NEXTR pointer points to the next routing record subordinate to the jobs record. The NEXTR pointer links the Routing records in ascending order by the routing record SERVER_GROUP_R field.

SERVICE_DIST. Integer. The BUILD_LL_FROM_DB procedure reads the value of the data base Routing record Service Distribution field into the dynamic job type record SERVICE_DIST field. Figure 2.4 lists the three distribution types and their integer codes.

SERVICE_RATE. Integer. The BUILD_LL_FROM_DB procedure reads the value of the data base Routing record Service Distribution Parameter field into the dynamic job type record SERVICE_RATE field. See Figure 2.4 for a description of the distribution parameter field.

ROUTING_PROB. Array (.1..10.) of Integer. The BUILD_LL_FROM_DB procedure reads the Routing Routing Probability array into the dynamic routing record ROUTING_PROB array.

3. Job Record

JOB_NUM. Integer. The job number is assigned to each job sequentially by the Create Job Stream Module as it enters the arrival queue.

NEXT_JOB. Pointer. The Next Job pointer points to the next job record in the queue. NEXT_JOB links jobs in the arrival, server group, and exit queues.

FIRST_JOB_PART. Pointer. Points to the first event record of the job.

ARRIVAL_TIME. Integer. The time the job arrives in the system, relative to the starting time of the simulation run. The arrival time is calculated by the Create Job Stream Module. It is the sum of a random variate which represents the job interarrival time plus the arrival time of the previous job of the same job type.

EXIT_SYS_TIME. Integer. Time job exits system, relative to the start time of the simulation. The exit system time value is entered into the record by the ARRIVE_At_SG procedure of the Execute and Tabulate Module.

PRIORITY. Integer. Priority value is entered by the CREATE_JOB procedure when the job is created. PRIORITY value is assigned from the PRIORITY_JT field of the dynamic job type record corresponding to the job type number of the job being created.

TIME_IN_SYS. Integer. Time in system is the difference between the EXIT_SYS_TIME and the ARRIVAL_TIME. The ARRIVE_AT_SG procedure of the Execute and Tabulate Module calculates TIME_IN_SYS when the job has completed processing. The calculation equation is presented in equation 4.1.

TIME_IN_SYS:= EXIT_SYS_TIME - ARRIVAL_TIME (eqn 4.1)

PROCESSING_TIME. Integer. Sum of the actual service times of all events comprising the job. Sum of the TIME_JOB_PART_TAKES fields in the job event records. The Create Job Stream module calculates the PROCESSING_TIME after all the job events have been created.

JOB_TYPE. Integer. The CREATE_JOB procedure enters the JOB_TYPE value when the job record is created. The JOB_TYPE value is assigned from the JOB_TYPE_JT field of the dynamic job type record corresponding to the job type number of the job being created.

TIME_IN_QUEUE. Integer. The ARRIVE_AT_SG procedure of the Execute and Tabulate Module calculates the job TIME_IN_QUEUE when the job processing is complete. The PASCAL calculation equation is presented in equation 4.2.

TIME_IN_QUEUE:= TIME_IN_SYS - PROCESSING_TIME (eqn 4.2)

SERVING_EVENT. Pointer. The Serving Event pointer points to the event record in the job which is currently being processed. Every time a job departs from a server group, the program updates the SERVING_EVENT pointer to point to the next event in the job.

4. Event Record

JOB_PART_NO. Integer. The Job Part Number identifies the events which comprise the job. The Create Job Stream Module assigns JOB_PART_NO to the job event records in sequential order commencing with '1' at the time the job events are created.

NEXT_JOB_PART. Pointer. The Next Job Part pointer points to the next sequential event record subordinate to the Job Record. The Event Records are linked by the Next Job Part pointer in ascending order by JOB_PART_NO at the time of job creation. Once created, the pointers remain unchanged for the duration of the simulation run.

SERVER_GROUP_NO. Integer. The Server Group Number identifies the server group at which the job event processing takes place.

TIME_JOB_PART_TAKES. Integer. The Time Job Part Takes indicates the processing time for the job event. The processing time is a random variate calculated by the Create Job Stream Module from the Service Distribution Type and Service Distribution Parameter of the Routing Record corresponding to the server group at which the job event is being processed.

5. Server Group Record

SERVER_GROUP. Integer. Server Group Number valid range is 0 to 10. The server group number uniquely identifies the server group in the simulation and corresponds to the input parameters created by the user in the Server Group Record.

NEXT_SERVER_GROUP. Pointer. The Next Server Group pointer points to the next server group in the simulation in sequential order by Server Group Number. The Server Group Records are linked in sequential order when they are created by the Create Server Group Procedure. Once created, the sequential order of the server groups and the value of the Next Server Group pointers remains unchanged for the duration of the program execution.

FIRST_SERVER. Pointer. The First Server points to the Server Record of the first server in the Server Group.

FIRST_IN_Q. Pointer. Points to the Job Records. For Server Group 0 the First In Queue pointer points to the arrival job queue, the linked list of job and event records that define jobs waiting to enter the system. For Server Groups 1 to 9, the First In Queue pointer points to the first job in the queue waiting for service at the server group. If there is no queue, the FIRST_IN_Q value is nil.

Q_LEN_VECTOR. Integer. For Server Groups 1 through 9, the Queue Length Vector is incremented every time the queue length changes by adding the queue length multiplied by the amount of time the queue was the indicated length. The PASCAL formula is presented in equation 4.3.

```
Q_LEN_VECTOR:= Q_LEN_VECTOR                          (eqn 4.3)
+ (( CLK - Q_LEN_TIME) * Q_LENGTH)
```

CLK is the mnemonic for the Clock variable used during the simulation run to indicate current simulation run time.

Q_LEN_TIME. Integer. The Queue Length Time indicates the clock time that the server group queue became the length indicated in the Q_LEN field.

Q_LEN. Integer. The Queue length indicates the current length of the queue. If there is no queue at the server group, the Q_LEN variable is assigned a value of '0'.

MAX_Q_LENGTH. Integer. The Maximum Queue Length contains the value corresponding to the maximum length of the server group queue since the beginning of the simulation run.

CUM_BUSY_TIME. Integer. The Cumulative Busy Time is the sum of the times that the server group has been busy since the beginning of the simulation run. The Cumulative Busy Time is incremented every time the server group goes from a busy to an idle status, or at the end of the simulation run. The PASCAL formula for the Cumulative Busy time computation is given in equation 4.4.

```
CUM_BUSY_TIME:= CUM_BUSY_TIME                    (eqn 4.4)
+ (CLK - START_BUSY_TIME)
```

MIN_Q_LENGTH. Integer. The Minimum Queue Length contains the value corresponding to the minimum length of the server group queue since the beginning of the simulation run.

START_BUSY_TIME. Integer. If the server group is busy then the Start Busy Time contains the clock time when the server group last went from an idle to a busy status. If the server group is idle, the Start Busy Time value is '0'.

NEXT_MAST_Q. Pointer. The Next In Master Queue pointer is used to link the server group records in the Master Event

Queue. The records are linked in ascending order by NEXT_EVENT_TIME. Server group records of idle server groups have nil values for the Next In Master Queue pointer and are not included in the Master Event Queue linked list.

NEXT_EVENT_TIME. Integer. For a busy server group, the Next Event Time field indicates the clock time that the next event in the server group will finish processing. The Next Event Time is the lowest of the TIME_EXIT fields of the busy servers (server records) in the server group. If the server group is idle, the Next Event Time value is '0'.

NEXT_S_EVENT. Pointer. The Next Server Event pointer points to the server record in the server group at which the next event will be completed. This is the server record with the lowest TIME_EXIT value which corresponds to the NEXT_EVENT_TIME in the server group record.

6. Server Record

SERVER. Integer. The Server field is the identifying number of the server in the server group. The Server number is assigned in sequential order beginning with '1' by the CREATE_SERVER_GROUP procedure when the server records are created.

NEXT_SERVER. Pointer. The Next Server pointer points to the next sequential server record in the server group. The server records are linked in sequential order by the Next Server pointer at the time they are created by the CREATE_SERVER_GROUP procedure. The order and value of the Next Server pointers do not change for the duration of the simulation run.

S_START_BUSY. Integer. If the server is busy then the Server Start Busy Time contains the clock time when the server last went from an idle to a busy status. If the server is idle, the Server Start Busy Time value is '0'.

TIME_EXIT. Integer. If the Server Group is busy, the Time_Exit value is the clock time that the job event will complete processing at the server group. It is calculated by the ARRIVE_AT_SG procedure when a job is attached to an available server. The TIME_EXIT value is the sum of the the current clock time (CLK) plus the TIME_JOB_PART_TAKES field of the job event being processed. If the Server Group is idle, the value of TIME_EXIT is '0'.

S_CUM_BUSY. Integer. The Server Cumulative Busy Time is the sum of the times that the server has been busy since the beginning of the simulation run. The Server Cumulative Busy Time is incremented every time the server goes from a busy to an idle status. The PASCAL formula calculation is presented in equation 4.5.

```
S_CUM_BUSY:= S_CUM_BUSY                                    (eqn 4.5)
+ (CLK - S_START_BUSY)
```

JOB_NO. Pointer. The Job Number pointer points to the job record of the job being serviced by the server. If the server is idle, the value of JOB_NO is nil.

BUSY. Boolean. The BUSY field is true if the server is busy and false if the server is idle.

G. FILE DESCRIPTION AND MAINTENANCE

The physical files which comprise the CPMT are listed in Figure 4.4.

1. Pascal Source Files

The logical procedures and modules which make up the source code of the CPMT are divided into the physical Pascal files as indicated in Figure 4.4. The source code is kept in separate files roughly corresponding to the logical

Pascal Source Files:	
CPMT.PAS	CPMT Main Driver
UPMOD.PAS	Data Base Update Module
CHECKSS.PAS	Check Simulation Specifications Module
CJS.PAS	Create Job Stream Module
EXT.PAS	Execute and Tabulate Module
Data Files:	
RECFILE.DAT	Indexed Sequential Data Base
MESSAGES.DAT	File of Dialogue Messages
OUTFILE.DAT	CPMT Output File

Figure 4.4 CMPT Physical Files.

program modules for ease of development, maintenance and testing of the program. The file CPMT.PAS is the main program driver procedure, and contains the '%INCLUDE' directive which calls in the component source code files during compilation. To change the PASCAL program, the programmer makes the change in the source file in which the module code resides, and then recompiles the file CPMT.PAS.

2. Data Files

RECFILE.DAT contains the indexed sequential data base which is updated by the CPMT Update Module. RECFILE.DAT file maintenance and organization is described in Chapter 5.

MESSAGES.DAT is a sequential file which contains text messages that CPMT can send to an output file or terminal. The program uses the MESSAGES.DAT file for many of the interactive dialogue messages and terminal displays. The file messages are identified by a message number and

delimited by the '$' character. When the CPMT program accesses the file, it reads sequentially through the file until it encounters the desired message number. It then transfers the corresponding text message contained between the '$' signs to the specified output device or file. The programmer can change the MESSAGES.DAT file using the system editor. A change to the file does not require recompilation of the CPMT program.

OUTFILE.DAT is a sequential output file to which the CPMT program writes data base printouts, simulation specification check error messages, and the simulation execution statistical output reports. The CPMT program creates a new OUTFILE.DAT for each terminal session.

V. DATA BASE SPECIFICATIONS

The CPMT data base of simulation model specifications is implemented as an indexed sequential data base using the VAX-11 RMS (Record Management Services). The data base is organized into four logical record structures: the header record, the server group record, the job type record and the routing record. The four logical record types have a common physical structure.

A. DATA BASE LIMITATIONS

The data base is designed to accommodate the specifications of 99 different simulation models. The limiting factor is the designated maximum simulation number used in calculating the record keys. Each simulation model specification requires one header record, one server group record, from 1 to 99 job type records, and from 2 to 10 routing records for each job type record.

B. DATA BASE UPDATE AND ACCESS

The data base is updated interactively under control of the CPMT program Update Module. The data base update options include: adding and deleting server group, job type and routing records; copying an entire simulation model to a new simulation model number; and deleting a simulation model from the data base.

C. RECORD KEY

The indexed sequential record key is an integer value which is calculated from one to four component parameters,

depending on the record type. The four parameters are: the simulation model number, the job type number, the record type number and the routing record server group number. The record key value determines the logical ordering of the records in the data base. The key for the CPMT data base is designed so that the simulation model number records are sequentially grouped. For a given simulation model, the header record is first and the server group record is second, followed by the first job type record and its associated routing records, then the second job type record and its associated routing records, and so on depending on the number of job types in the model. The record key component fields are presented in Figure 5.1 and the record key calculations for each record type are presented in Figure 5.2.

NAME	MNEMONIC	RANGE
Simulation Model Number	SIMNUM	1..99
Job Type Number	JOBNUM	1..99
Record Type	RECTYPE	0 - Header Rec 1 - Job Type Rec 2 - Routing Rec 3 - Server Group Rec
Routing Record Server Group Number	RR_S_NUM	0..9

Figure 5.1 Record Key Components.

RECTYPE	RECORD	CALCULATION
0	Header	(SIMNUM * 100000)
1	Job Type	(SIMNUM * 100000) (JOBNUM * 1000) (RECTYPE * 100)
2	Routing	(SIMNUM * 100000) (JOBNUM * 1000) (RECTYPE * 100) (RR_S_NUM * 1)
3	Server Group	(SIMNUM * 100000) (RECTYPE * 100)

Figure 5.2 Record Key Calculations.

D. LOGICAL AND PHYSICAL RECORD STRUCTURES

The four logical record types share a common physical record description. The physical record structure is defined as type DB_RECORD (Data Base Record) in the TYPE section of the main driver CMPT program. The DB_RECORD fields and the corresponding mapping of the logical record fields for the four logical record types are shown in Figure 5.3 through Figure 5.6. The description of the logical record fields and their valid data ranges is presented in Chapter 2.

DB_RECORD FIELDS	FIELD TYPE	HEADER RECORD FIELDS
RECORD_KEY	INTEGER	RECORD KEY
REC_RATE	INTEGER	** Not Used **
REC_DIST	INTEGER	** Not Used **
REC_PRIORITY	INTEGER	** Not Used **
REC_DESC	VARYING {50} OF CHAR	** Not Used **
REC_ARRAY	ARRAY {1..10} OF INTEGER	** Not Used **

Figure 5.3 Header Record Field Correspondence.

E. RMS/PASCAL COMMANDS

The Update Module uses the following RMS/PASCAL file management commands to control access and update of data base records:

* OPEN - Used to open the data file. Calling parameters include the file name, history, organization and access method.
* CLOSE - Used to close the data file. Calling parameters include the file name.
* FINDK - Used to randomly access a record in the data file. File pointer is positioned to the record indicated by the record key and the record is available for program access if found. Calling parameters include the file name, key offset, and record key.

DB_RECORD FIELDS	FIELD TYPE	SERVER GROUP RECORD FIELDS
RECORD_KEY	INTEGER	RECORD KEY
REC_RATE	INTEGER	** Not Used **
REC_DIST	INTEGER	** Not Used **
REC_PRIORITY	INTEGER	** Not Used **
REC_DESC	VARYING {50} OF CHAR	** Not Used **
REC_ARRAY	ARRAY {1..10} OF INTEGER	NUMBER SERVERS

Figure 5.4 Server Group Record Field Correspondence.

* WRITE - Used to write a record to the data base. Calling parameters include file name and variable name of record that is being written to the file.

* GET - Used to advance the file pointer to the next logically consecutive record in the file. Used for sequential access of the data records. After executing a get command the data record is available in a file buffer and may be read from the buffer into a program defined record variable for access. Calling parameters include the file name.

* RESETK - Used to reset the file pointer to the beginning of the file. Calling parameters include file name and key number parameters.

* DELETE - Used to delete the record currently pointed to by the file pointer. Before the delete command is

DB_RECORD FIELDS	FIELD TYPE	JOB TYPE RECORD FIELDS
RECORD_KEY	INTEGER	RECORD KEY
REC_RATE	INTEGER	ARRIVAL DIST PARAMETER
REC_DIST	INTEGER	ARRIVAL DIST
REC_PRIORITY	INTEGER	JOB TYPE PRIORITY
REC_DESC	VARYING {50} OF CHAR	** Not Used **
REC_ARRAY	ARRAY {1..10} OF INTEGER	** Not Used **

Figure 5.5 Job Type Record Field Correspondence.

executed it is necessary to position the pointer to the desired record with a FINDK or GET command. Calling parameters include parameters for file name.

* UPDATE - Used to rewrite a record in the data base. Calling parameters include file name.

F. DATA BASE FILE MAINTENANCE

The indexed sequential data base is located in file RECFILE.DAT. The CPMT program will automatically access the file RECFILE.DAT in the directory in which the program is executing. If the program does not find the data base file in its directory, it creates a new indexed sequential file named RECFILE.DAT in that directory.

DB_RECORD FIELDS	FIELD TYPE	ROUTING RECORD FIELDS
RECORD_KEY	INTEGER	RECORD KEY
REC_RATE	INTEGER	SERVICE DIST PARAMETER
REC_DIST	INTEGER	SERVICE DIST
REC_PRIORITY	INTEGER	** Not Used **
REC_DESC	VARYING {50} OF CHAR	** Not Used **
REC_ARRAY	ARRAY {1..10} OF INTEGER	ROUTING PROBABILITY

Figure 5.6 Routing Record Field Correspondence.

The automatic file creation characteristic of the data base has the following implications for users and maintainers of the data base:

- To initialize the data base, it is sufficient to delete the current copy of RECFILE.DAT and have the CPMT program recreate the file during the next program execution. If the CPMT program is copied to and run under a new directory, the CPMT user for that directory can either copy an existing RECFILE.DAT data file into their directory or have the CPMT program create a new file.
- If the data base record structure type DB_RECORD is changed, the CPMT program will not run against a data base file created before the change. If not concerned about data loss, the user can delete the existing data base and have the program create a new file to accommodate the changed record structure.

VI. TEST AND VERIFICATION

In order to verify the CPMT program, simulation models which could be analytically solved were developed and run using CPMT. The simulation model results and analytical model solutions are compared below for two simple computer system models extracted from [Ref. 1: pp. 167 - 174].

A. TEST MODEL #1

The first test model is the model of a single server computer with a single job type. Jobs arrive into the system at a rate of 10 jobs per hour and have a mean service time of 3 minutes. The CPMT simulation parameters developed for Test Model #1 are displayed in Figures 6.1 and 6.2.

Simulation Number: 1

Server Group Number:	Number Servers:
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Figure 6.1 Server Group Parameters for Test Model #1.

Simulation Number: 1 Job Type Number: 1

********** Job Type Record Data **********

Arrival Dist: 2

Dist Param: 6

Job Type Priority: 1

********** Routing Record Data **********

Server Group:	0	1	2	3	4	5	6	7	8	9
Service Dist:	NA	2								
Dist Param:	NA	3								
Routing To:										
SG 1	100	0								
SG 2	0	0								
SG 3	0	0								
SG 4	0	0								
SG 5	0	0								
SG 6	0	0								
SG 7	0	0								
SG 8	0	0								
SG 9	0	0								
SG 10	0	100								

Figure 6.2 Job Type and Routing Parameters for Test Model #1.

The analytical solution of the model taken from [Ref. 1] indicates that utilization of the server is .50 and that the response time of the job is 6 minutes. The analytical model results are compared to the results of 10 independent simulation model runs in Figure 6.3. The simulation runs are of different lengths based on the number of jobs run in the simulation and use different seed values.

TEST DATA: TEST #1

SIMULATION NUMBER: 21

			Util 1	Resp Time	Stdd Rtime
ANALYTIC RESULTS:			.50	6.00	
SIMULATION MODEL RESULTS:					
Attempt	Num Jobs	Seed			
1.1	200	987255	.531	5.690	4.549
1.2	400	99987	.547	6.255	4.721
1.3	300	53726	.555	6.300	5.948
1.4	400	53726	.529	6.153	5.546
1.5	250	9675	.540	7.040	6.095
1.6	300	75439	.513	5.733	5.091
1.7	1000	29983	.534	5.902	4.835
1.8	700	889203	.571	5.764	4.702
1.9	900	299853	.514	6.258	5.554
1.10	500	47309	.527	6.268	5.554

Figure 6.3 Test #1 Data.

B. TEST MODEL #2

The second test model is the model of a three server group computer system with a single job type. The analytical model results from [Ref. 1] indicate a response time of 324 for the jobs and utilization percentages of .27 for Server Group 1; .20 for Server Group 2; and .50 for Server Group 3. The CPMT simulation parameters for the model are developed in Chapter 2 and listed in Figures 2.11 and 2.12. The analytical model results are compared to the results of 10 independent simulation model runs in Figure 6.4.

C. HYPOTHESIS TESTING OF RESPONSE TIME MEANS

As a method of verifying the CPMT simulation results, the simulation run response time means are compared to the analytically calculated response time means with the intention of determining whether the means come from the same population. For each simulation run, we establish the null hypothesis that the population mean is equal to the sample mean, where the analytical response time is designated as the population mean and the simulation result as the sample mean. The alternative hypothesis is that the analytical mean does not equal the simulated mean. We then test the hypothesis at a level of significance of a = .05 and a = .01 by computing the test statistic using the formula presented

$$Z = (X - u) / (S / \sqrt{N}) \qquad \text{(eqn 6.1)}$$

in equation 6.1. In the equation X is the sample run response time mean; u is the analytical or population mean; S is the standard deviation of the sample run response time mean; and N is the number of jobs run in the sample.

TEST DATA: TEST #2

SIMULATION NUMBER: 25

			Jtil 1	Util 2	Util 3	Resp Time	Stdd Rtime
ANALYTIC RESULTS:			.27	.20	.50	324	
SIMULATION MODEL RESULTS:							
Attempt	Num Jobs	Seed					
2.1	100	4325	.319	.280	.543	390	421
2.2	500	89763	.294	.226	.522	356	420
2.3	1000	89274	.268	.193	.471	323	428
2.4	3000	78653	.27	.190	.477	290	338
2.5	9000	78653	.277	.197	.494	314	377
2.6	9000	75983	.281	.201	.497	318	373
2.7	9000	389254	.282	.199	.496	317	388
2.8	800	33333	.271	.201	.505	336	418
2.9	3000	33333	.270	.194	.484	317	402
2.10	9000	4453	.279	.201	.493	328	393

Figure 6.4 Test #2 Data.

At the .01 level of significance, if the test statistic is greater than 2.580 the null hypothesis is rejected, otherwise it is not rejected. At the .05 level of significance, the null hypothesis is rejected if the test statistic is greater than 1.960. Figure 6.5 and 6.4 present the test statistic calculations and the hypothesis decisions for Test Model #1 and Test Model #2 respectively.

Attempt	Z Calc for Rtime Mean	Level of Significance a = .05 Reject H?	a = .01 Reject H?
1.1	.969	No	No
1.2	1.08	No	No
1.3	.874	No	No
1.4	.551	No	No
1.5	2.698	Yes	Yes
1.6	.908	No	No
1.7	.641	No	No
1.8	1.317	No	No
1.9	1.394	No	No
1.10	1.079	No	No

Figure 6.5 Test #1 Hypothesis Test of Rtime Mean.

D. CONCLUSIONS

The hypothesis testing of the Test Model #1 response time means indicates that for 9 out of 10 samples there is no statistically significant difference between the analytic mean and the simulation mean. The hypothesis testing for the Test Model #2 response time means indicates that for 9 out of 10 samples there is no statistically significant difference between the analytic mean and the simulation mean at the .01 level of significance. At the .05 level of

Attempt	Z Calc for Rtime Mean	Level of Significance a = .05 Reject H?	a = .01 Reject H?
2.1	1.568	No	No
2.2	1.704	No	No
2.3	.074	No	No
2.4	5.51	Yes	Yes
2.5	2.516	Yes	No
2.6	1.526	No	No
2.7	1.712	No	No
2.8	.812	No	No
2.9	.954	No	No
2.10	.966	No	No

Figure 6.6 Test #2 Hypothesis Test of Rtime Mean.

significance for Test Model #2, there is no statistically significant difference in the means for 8 out of 10 samples.

VII. CONCLUSIONS

The CPMT baseline program is operational and has been used as part of a class exercise for CS4400. The program validation and test results discussed in Chapter 6 indicate that the simulation results are accurate at a level of significance acceptable for the purposes of the program.

Consistent with the goal that the CPMT baseline program be used as a basis for ongoing simulation program enhancement and as a tool for CS4400, the following list of program enhancement possibilities is presented. The list is culled from the comments of CS4400 class members, the initial program users, and from features included in the original program design which have not yet been implemented due to time constraints.

CPMT Enhancement possibilities:

1. The CPMT server group queueing discipline is currently first come, first served. Queueing discipline possibilities can be extended to include other queueing disciplines. In addition, a round robin or time slice processing algorithm can be implemented for appropriate server groups.
2. The CPMT simulation run duration is currently specified by the number of jobs run through the modeled system. An alternative means of specifying run duration is by length of time based on the Execute and Tabulate Module clock. If the latter capability is implemented, the user could be given the option of specifying duration based upon number of jobs or simulation execution clock time.
3. The CPMT program currently does not provide a method for overcoming the statistical bias introduced by the

execution start up and shut down transition phases when jobs are starting to come into the system and when all jobs are leaving the system. The program could be enhanced to allow the user to specify an interval during which statistics are gathered. The interval could be specified in terms of the execution clock or number of jobs. For example, the user could specify that the simulation is to run for 1000 time units and that system statistics are to be determined for the 100 to 1000 time interval in order to avoid statistical gathering during the start up transition phase.

4. The job and event records which describe the jobs processed by the simulation run are all created before the program starts to process jobs through the simulated system. The dynamic job records are 'kept' when they exit the simulated system. When all the jobs are processed through the system, the program traverses the list of completed job records to calculate the job type statistics. The problem with having all the job records in the system is that for sizable simulation runs, the computer system limitations are reached. The possibility of gathering job statistics as the jobs exit the system and then releasing the job records could be investigated as a program enhancement. The program logic can easily be changed so that the program creates the jobs as they enter the simulated system instead of creating all jobs before the simulation execution begins. In order to do that, the CREATE_JOB procedure can be called from the main loop of the EXECUTE_AND_TABULATE module when a new job needs to be placed in the arrival queue.

5. The CPMT program currently writes the data base printout, the check simulation specification error messages, and the simulation run statistical report to a single output file which is newly created for each execution session. The program could be changed so that the three different types of output are written to different files.
6. Other enhancements to increase the user friendliness of the CPMT program include: an option in the UPDATE module which would display the used/unused simulation model numbers in the data base; an option to print the data base specifications for a single model number as well as for the data base as a whole; options to change simulation model data base records in addition to the addition and deletion options.

APPENDIX A

CPMT PASCAL SOURCE CODE

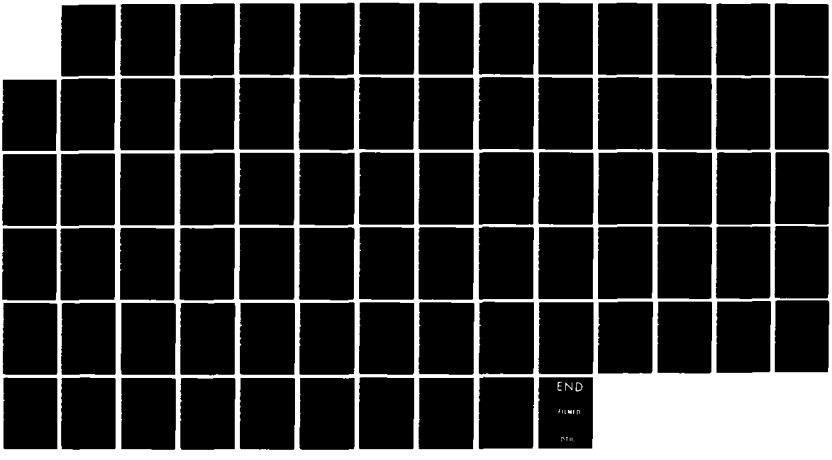
```
(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
(*                    CPMT PROGRAM                       *)
(*                                                       *)
(* THE COMPUTER PERFORMANCE MODELING TOOL (CPMT)         *)
(* IS A DISCRETE EVENT SIMULATION PROGRAM DESIGNED TO    *)
(* MODEL COMPUTER SYSTEMS.   THE CPMT PROGRAM CONSISTS   *)
(* OF THE MAIN DRIVER, THE UPDATE MODULE, THE CHECK      *)
(* SIMULATION SPECIFICATIONS MODULE, THE CREATE JOB      *)
(* STREAM MODULE, AND THE EXECUTE AND TABULATE MODULE.   *)
(*-------------------------------------------------------*)

  PROGRAM CPMT

 ( INPUT, OUTPUT, RECFILE, MESSAGES, OUTFILE, );
  CONST

    MAIN_MEN              = 15;
    MAIN_MENU             = 41;
    ADD_JT                =  2;
    ADD_RR                =  3;
    ADD_SERV              =  4;
    DEL_JT                =  5;
    DEL_RR                =  6;
    DEL_SERV              =  7;
    CHG_JT                =  8;
    COP_SIM               =  9;
    DEL_SIM               = 10;

    ENT_SIM_NUM           = 11;
    SIM_NUM_IS            = 12;
    UPD_MENU              = 13;
    JT_NUM_IS             = 14;

    JT_DIST               = 24;
    JT_PRIORITY           = 25;
    JT_LABEL              = 26;

    RR_JOB_NUM            = 32;
    RR_SERV_NUM           = 33;
    RR_DIST               = 34;
    RR_PROB_ERR           = 28;
```

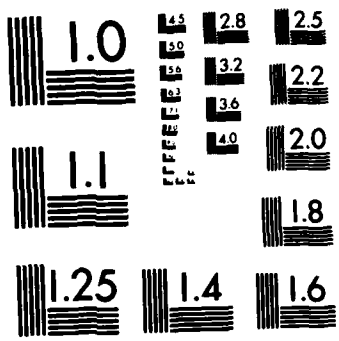
```
JT_DEL_NUM              = 30;
JT_DEL_ERR              = 31;
TO_PROB                 = 32;

SVCGPPRIOR              = 33;
JBPRIOR                 = 34;

SVCGPWINIT              = 35;
JBINIT                  = 36;

SEEDPAR                 = 19;
NUM_MSG                 = 43;
SIMPAR                  = 42;

OUTP_HEAD               = 50;
JOB_HEAD                = 51;
ARRIVAL_RT_HD           = 52;
ARRIVE_AT_HD            = 53;
ROUTE_HEAD              = 54;
FROM_HD                 = 55;
TO_HD                   = 56;
FROM_PROB_HD            = 57;
TO_PROB_HD              = 58;
ROUTE_SEP               = 59;
GOBACK                  = 60;

MENUE3                  = 61;
DONE                    = 62;
ASTERISK                = 63;
ALERT                   = 64;

INIT_MEN                =  1;

MAX_SERV                = 10;
MAX_SIM                 = 99;
MAX_RTYPE               =  3;


MAX_RNUM                = 99;
MAX_PRIORITY            = 10;
MAX_DIST                =  3;

MAX_RATE          = 999999;
MAX_QUE_DISP            =  4;

HD_REC                  =  0;
```

```
   JT_REC                  =  1;
   RK_REC                  =  2;
   SG_REC                  =  3;


TYPE


DB_RECORD = RECORD
          RECORD_KEY:  KEY (0)  INTEGER;
          REC_RATE:  INTEGER;
          REC_DIST:  1 .. MAX_DIST;
          REC_PRIORITY:  1 .. MAX_PRIORITY;
          REC_DESC:  VARYING  50  OF CHAR;
          REC_ARRAY:  ARRAY  1..MAX_SERV  OF  INTEGER;

END; (* RECORD TYPE *)


   NODE  =  (SERVER_GROUP_RECORD, SERVER_RECORD,
             Q_RECORD, JOB_RECORD, EVENT_RECORD,
             JOBS, ROUTING);

   PTR = @PERFORMANCE_RECORD;
   PERFORMANCE_RECORD = RECORD
     CASE TAG : NODE OF

            SERVER_GROUP_RECORD:
              (SERVER_GROUP:INTEGER;
               FIRST_SERVER:PTR;
               NEXT_SERVER_GROUP:PTR;
               FIRST_IN_Q:PTR;
               Q_LENGTH : INTEGER;
               Q_LEN_VECTOR : INTEGER;
               Q_LEN_TIME : INTEGER;
               MAX_Q_LENGTH :INTEGER;
               MIN_Q_LENGTH: INTEGER;
               START_BUSY_TIME: INTEGER;
               STOP_BUSY_TIME : INTEGER;
               CUM_BUSY_TIME:INTEGER;
               NEXT_MASTQ: PTR;
               NEXT_EVENT_TIME: INTEGER;
               NEXT_S_EVENT: PTR);

         SERVER_RECORD:
               (SERVER: INTEGER;
```

AD-A154 438 COMPUTER PERFORMANCE MODELING TOOL (CPMT)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA K A PAGEL DEC 84 2/2

UNCLASSIFIED F/G 9/2 NL





```
            JOB_NO: PTR;
            S_START_BUSY :INTEGER;
            S_CUM_BUSY: INTEGER;
            TIME_EXIT: INTEGER;
            NEXT_SERVER: PTR;
            BUSY: BOOLEAN);

    JOB_RECORD :
         (JOB_NUMB : INTEGER;
          ARRIVAL_TIME : INTEGER;
          EXIT_SYS_TIME : INTEGER;
          FIRST_JOB_PART : PTR;
          SERVING_EVENT: PTR;
          NEXT_JOB: PTR;
          PRIORITY: INTEGER;
          TIME_IN_SYS: INTEGER;
          PROCESSING_TIME: INTEGER;
          JOB_TYPE: INTEGER;
          TIME_IN_QUEUE : INTEGER);

    EVENT_RECORD :
         (JOB_PART_NO : INTEGER;
          SERVER_GROUP_NO : INTEGER;
          TIME_JOB_PART_TAKES : INTEGER;
          NEXT_JOB_PART : PTR;
          SCHEDULED: BOOLEAN);


    JOBS:
         (JOB_TYPE_JT : INTEGER;
          ARRIVAL_RATE : INTEGER;
          ARRIVAL_DIST : INTEGER;
          PRIORITY_JT : INTEGER;
          NEXTJ : PTR;
          ROUTINGJ : PTR );

    ROUTING:
         (SERVER_GROUP_R : INTEGER;
          SERVICE_DIST : INTEGER;
          SERVICE_RATE : INTEGER;
          ROUTING_PROB : ARRAY (.1..10.) OF INTEGER;
          NEXTR : PTR );

END;     (* RECORD TYPE *)

WORD_INTEGER =  WORD  0..65535;
```

```
VAR

  REVERSE: WORD_INTEGER:= 2;
  RECFILE : FILE OF DB_RECORD;
  DREC: DB_RECORD;
  SCREEN_STAT, LINE, COLUMN, COUNTER : INTEGER;
  I: INTEGER;
  OUTFILE:TEXT;                                    (* OUTPUTFILE *)
  MESSAGES:TEXT;            (* FORMATTED INFORMATION FILE *)
  TEST:INTEGER;                                    (* TEST VALUE *)
  DB_LIST: TEXT;
  SVC_GROUPTR:PTR;  (* POINTER TO SEVICE_GROUPS RECORDS *)
  JOB_TYPTR:PTR;                (* POINTER TO JOBS RECORDS *)
  SEED:INTEGER;    (* USED IN RANDOM NUMBER FUNCTION    *)
  ARRIVEQPTR:PTR;
  FIRST_SERVER_GROUP_PTR : PTR;
  PRINT: BOOLEAN;
  NUM_SERVER_GROUPS : INTEGER;
  SAVE_SIMNUM:INTEGER;
  HIGHEST_JOB_TYPE: INTEGER;
  NUM_SERVERS : ARRAY  1..10  OF INTEGER;

  MAIN_OPT: INTEGER;
  EXIT_MAIN: BOOLEAN;

  NUMJOBS:INTEGER;
  JOBPNUM:INTEGER;

  SIMNUM: INTEGER;
  JOBNUM: INTEGER;
  RR_S_NUM: INTEGER;
  RECTYPE: INTEGER;
  ANS: CHAR;

  SIMCHECK: BOOLEAN;

EXTERNAL  FUNCTION LIB$ERASE_PAGE
 (LINE_NO: WORD_INTEGER;
  COL_NO : WORD_INTEGER) : INTEGER; EXTERN;


EXTERNAL  PROCEDURE LIB$STOP
   (%IMMED COND_VALUE : INTEGER);
    EXTERN;
```

```
{*-------------------------------------------------------*}
{*-------------------------------------------------------*}
{*                  UPDATE MODULE                        *}
{*-------------------------------------------------------*}
{*-------------------------------------------------------*}


{*-------------------------------------------------------*}
{*                                                       *}
{* FIND_MSG: LOCATES A MESSAGE WITH THE MESSAGE NUMBER   *}
{*   PASSED TO THE VARIABLE MSG. MESSAGE IS IN THE       *}
{*   MESSAGE FILE MESSAGE.DAT.                           *}
{*                                                       *}
{*-------------------------------------------------------*}


  PROCEDURE FIND_MSG(VAR MESSAGES:TEXT; MSG:INTEGER);

    VAR C1:CHAR; DIGIT:INTEGER;

    BEGIN
      RESET (MESSAGES);
      C1:=' ';
      DIGIT:=-1;
      REPEAT
        READ(MESSAGES,C1);
        IF C1 = '$' THEN
          READLN(MESSAGES,DIGIT)
        ELSE
          READLN(MESSAGES);
      UNTIL ((C1='$') AND (DIGIT=MSG));
    END;                                          (* FIND_MSG *)
{*-------------------------------------------------------*}


{*-------------------------------------------------------*}
{* PRINT_MSG: PRINTS A SPECIFIC ONE LINE MSG             *}
{*   TO ANOTHER FILE. DOES NOT WRITELN TO FILE.          *}
{*   USE FOR LINE LABELS OF GENERATED DATA               *}
{*-------------------------------------------------------*}
  PROCEDURE PRINT_MSG(VAR OUTFILE,MESSAGES:TEXT;
                                                MSG:INTEGER);
    VAR C1:CHAR;
    BEGIN
      FIND_MSG(MESSAGES,MSG);
```

```
      READ(MESSAGES,C1);
      WHILE C1<>'$' DO
        BEGIN
          WRITE(OUTFILE,C1);
          READ(MESSAGES,C1);
        END;
    END;                                              (* PRINT_MSG*)
(*-------------------------------------------------------------*)


(*-------------------------------------------------------------*)
(* PRINTLN_MSG: PRINTS A SPECIFIC MSG FROM ONE FILE           *)
(*   TO ANOTHER FILE.  THIS ROUTINE WILL TAKE THE             *)
(*   WHOLE MESSAGE AND PRINT IT.  IT DOES A WRITELN           *)
(*   AT THE END OF THE PRINT                                  *)
(*   FORMAT FOR THE MESSAGE FILE IS $NUM.  INFORMATION        *)
(*   TO BE PRINTED HAS TO BE ON THE NEXT LINE.                *)
(*-------------------------------------------------------------*)
  PROCEDURE PRINTLN_MSG(VAR OUTFILE,MESSAGES:TEXT;
                                                    MSG:INTEGER);
    VAR C1:CHAR;
    BEGIN
      FIND_MSG(MESSAGES,MSG);
      READ(MESSAGES,C1);
      WHILE C1<>'$' DO
        BEGIN
          WRITE(OUTFILE,C1);
          WHILE NOT EOLN(MESSAGES) DO
              BEGIN
                READ(MESSAGES,C1);
                WRITE(OUTFILE,C1);
              END;
          READLN(MESSAGES);
          WRITELN(OUTFILE);
          READ(MESSAGES,C1);
        END;
    END;                                          (* PRINTLN_MSG *)
(*-------------------------------------------------------------*)


(*-------------------------------------------------------------*)
PROCEDURE CLEAR_SCREEN;

(* ERASE THE SCREEN *)
BEGIN
```

```
    SCREEN_STAT := LIB$ERASE_PAGE (LINE_NO := 1,
       COL_NO := 1);
    IF NOT ODD (SCREEN_STAT) THEN
       LIB$STOP (SCREEN_STAT);

END ; (* PROCEDURE CLEAR_SCREEN)

(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE COMPUTE_KEY:                                *)
(*   THIS PROCEDURE COMPUTES AN INTEGER VALUE TO BE USED *)
(*   AS THE INDEXED SEQUENTIAL RECORD KEY.  THE          *)
(*   KEY VALUE IS COMPUTED FROM FOUR COMPONENETS:        *)
(*   THE SIMULATION NUMBER, THE RECORD TYPE, THE JOB     *)
(*   NUMBER, AND THE SERVER GROUP NUMBER OF THE          *)
(*   ROUTING RECORD.                                     *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE  COMPUTE_KEY (RECTYPE:INTEGER);

BEGIN

  CASE RECTYPE OF

  0:  DREC.RECORD_KEY := (SIMNUM * 100000);

  1:  DREC.RECORD_KEY := (SIMNUM * 100000) +
                         (JOBNUM *   1000) +
                         (RECTYPE  *  100);

  2:  DREC.RECORD_KEY := (SIMNUM * 100000) +
                         (JOBNUM *   1000) +
                         (RECTYPE *   100) +
                         (RR_S_NUM *    1);

  3:  DREC.RECORD_KEY := (SIMNUM * 100000) +
                         (RECTYPE *   100);

  END; (* CASE RECTYPE *)

END; (* PROCEDURE COMPUTE_KEY *)
```

```
(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE DECOMPOSE_KEY:                              *)
(*   THIS PROCEDURE TAKES A RECORD KEY INTEGER           *)
(*   VALUE AND BREAKS IT DOWN INTO ITS COMPONENT PARTS   *)
(*   SO THAT THE PROGRAMMER MAY EASILY REFER TO THE      *)
(*   VALUES OF THE KEY COMPONENETS.                      *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE DECOMPOSE_KEY;

BEGIN

  SIMNUM := TRUNC (DREC.RECORD_KEY / 100000 );


  JOBNUM := TRUNC (DREC.RECORD_KEY / 1000 ) -
            (SIMNUM * 100);

  RECTYPE:= TRUNC (DREC.RECORD_KEY / 100 ) -
            ((SIMNUM * 1000 ) + (JOBNUM * 10));

  RR_S_NUM := DREC.RECORD_KEY -
              ((SIMNUM * 100000) + (JOBNUM * 1000) +
               ( RECTYPE * 100));

END; (* PROCEDURE DECOMPOSE KEY )

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE PRINT_DATA_BASE:                            *)
(*   THIS PROCEDURE READS SEQUENTIALLY THROUGH THE       *)
(*   ENTIRE INDEXED SEQUENTIAL DATA BASE IN FILE         *)
(*   RECFILE.DAT, FORMATS THE DATA BASE RECORDS, AND     *)
(*   WRITES THE FORMATTED RECORDS TO FILE OUTFILE.DAT    *)
(*                                                       *)
(*-------------------------------------------------------*)
```

```
PROCEDURE PRINT_DATA_BASE ;
VAR
SAVE_SIM_NUM: INTEGER;

BEGIN (* PROCEDURE PRINT DATA BASE*)
  SAVE_SIM_NUM:= 0;
  RESETK (RECFILE, 0);
  WHILE NOT EOF (RECFILE) DO
    BEGIN
      DREC := RECFILE↑;
      DECOMPOSE_KEY;
      IF SIMNUM <> SAVE_SIM_NUM THEN
        BEGIN
          PAGE(OUTFILE);
          SAVE_SIM_NUM := SIMNUM;
          WRITELN (OUTFILE, 'SIMULATION NUMBER: ', SIMNUM:3);
          WRITELN (OUTFILE);
        END; (* IF SIMNUM *)
      WRITELN (OUTFILE);

      CASE RECTYPE OF
      1:  (* JOB TYPE RECORD *)

          BEGIN (* JOB TYPE RECORD *)
            WRITELN (OUTFILE, 'JOB TYPE RECORD:', JOBNUM:3);
            WRITELN (OUTFILE, 'RECORD    ARRIVAL    ',
                'ARRIVAL   ');
            WRITELN (OUTFILE, 'KEY       DIST       ',
                'PARAM     PRIORITY');
            WRITELN (OUTFILE);
            WRITELN( OUTFILE, DREC.RECORD_KEY:7,
                    DREC.REC_DIST:5,
                    DREC.REC_RATE,
                    DREC.REC_PRIORITY);
            WRITELN (OUTFILE);


          END;
      2:  (* ROUTING RECORD *)
```

```
            BEGIN

              WRITELN (OUTFILE, 'ROUTING RECORD');
              WRITELN (OUTFILE,  'JOB     ',
                       'SERVER  SERVICE  SERVICE ',
                       '        ROUTING PROBABILITIES TO SG:');
              WRITE (OUTFILE,  'TYPE     ',
                       'GROUP   DIST     PARAM    ');
              FOR I:= 1 TO MAX_SERV DO
                WRITE (OUTFILE, I:4);
              WRITELN (OUTFILE);
              WRITELN (OUTFILE);
              WRITE (OUTFILE,
                     JOBNUM:5, RR_S_NUM:5,
                     DREC.REC_DIST, DREC.REC_RATE);

              FOR I:= 1 TO MAX_SERV DO
                WRITE (OUTFILE, DREC.REC_ARRAY I :5);
              WRITELN (OUTFILE);
            END;

          3:  (* SERVER RECORD *)

            BEGIN
              WRITELN (OUTFILE,'SERVER GROUP RECORD');
              WRITE (OUTFILE,  'RECORD KEY');
              FOR I:= 1 TO MAX_SERV DO
                WRITE (OUTFILE, I:6);
              WRITELN (OUTFILE);
              WRITE (OUTFILE, DREC.RECORD_KEY);
              FOR I:= 1 TO MAX_SERV DO
                WRITE (OUTFILE,DREC.REC_ARRAY I :6);
              WRITELN (OUTFILE);
            END;

          END; (* CASE RECTYPE OF *)

          GET (RECFILE);

        END; (* WHILE NOT EOF *)

      END; (* PROCEDURE PRINT DATA BASE *)

      (*-------------------------------------------------*)

      (*-------------------------------------------------*)
      (*                                                 *)
```

```
(*   FUNCTION NEXT_JT_NUM                                          *)
(*     THE FUNCTION LOCATES THE FIRST AVAILABLE                    *)
(*     UNUSED JOB NUMBER IN THE DATABASE FOR A SPECIFIC            *)
(*     SIMULATION MODEL NUMBER.  WHEN A JOB TYPE RECORD            *)
(*     IS ADDED TO THE DATABASE IT WILL AUTOMATICALLY              *)
(*     BE ASSIGNED THE NEXT AVAILABLE JOB NUMBER.                  *)
(*                                                                 *)
(*-----------------------------------------------------------------*)

FUNCTION NEXT_JT_NUM: INTEGER;

VAR   FOUND: BOOLEAN;
      RECTYPE: INTEGER;
      N_JOBNUM : INTEGER;

BEGIN

  FOUND:= FALSE;
  RECTYPE:= 1;
  JOBNUM := 1;

  REPEAT
    COMPUTE_KEY (RECTYPE);
    FINDK  (RECFILE, 0, DREC.RECORD_KEY);
    IF UFB (RECFILE) THEN
       FOUND:= TRUE
    ELSE
       JOBNUM:= JOBNUM + 1
  UNTIL FOUND;

  NEXT_JT_NUM := JOBNUM


END;    (* FUNCTION NEXT JOB TYPE *)
(*-----------------------------------------------------------------*)


(*-----------------------------------------------------------------*)
(*-----------------------------------------------------------------*)


(*-----------------------------------------------------------------*)
(*                                                                 *)
(* PROCEDURE UPDATE_MENU:  THIS IS THE MAIN PROCEDURE              *)
(*    OF THE UPDATE MODULE.  PROCEDURE UPDATE USES  A              *)
```

```
(*    CASE STATEMENT TO CONTROL THE EXECUTION OF THE      *)
(*    PROCEDURES WITHIN THE UPDATE_MENU SCOPE.            *)
(*    SEPARATE PROCEDURES EXIST FOR THE DIFFERENT         *)
(*    FUNCTIONS WHICH THE USER CAN REQUEST.               *)
(*--------------------------------------------------------*)

PROCEDURE UPDATE_MENU;

TYPE   INPUT_INTEGER_RANGE = 0..99;
       GOOD_RANGE_SET = SET OF INPUT_INTEGER_RANGE;



VAR    UPD_OPT:INTEGER;
       UM_STOP: BOOLEAN;
       UM_INPUT_ERR: BOOLEAN;
       UM_STR_INP: VARYING  10  OF CHAR;
       UM_CHAR: CHAR;
       JOB_NUM: INTEGER;
       UM_GOOD_JOBNUM: BOOLEAN;
       OK_SIMNUM: GOOD_RANGE_SET;
       OK_DIST_PARAM: GOOD_RANGE_SET;
       OK_DIST: GOOD_RANGE_SET;
       OK_PRIORITY: GOOD_RANGE_SET;
       OK_SG_NUM: GOOD_RANGE_SET;
       OK_QUE_DISP: GOOD_RANGE_SET;


(*--------------------------------------------------------*)


(*--------------------------------------------------------*)
(*                                                        *)
(*  PROCEDURE INITIALIZE_REC:                             *)
(*    THIS PROCEDURE INITIALIZES ALL THE FIELDS IN        *)
(*    THE VARIABLE DREC.                                  *)
(*                                                        *)
(*-------------------- ----------------------------------*)

PROCEDURE INITIALIZE_REC;

BEGIN
```

```
    WITH DREC DO
      BEGIN
        RECORD_KEY := 0;
        REC_RATE := 0;
        REC_DIST := 0;
        REC_PRIORITY := 0;
        REC_DESC:= '      ';
        FOR I:= 1 TO MAX_SERV DO
          REC_ARRAY I  := 0;
      END;

  END; (* PROCEDURE INITIALIZE RECORD     *)

  (*-------------------------------------------------------------*)

  (*-------------------------------------------------------------*)
  (*                                                             *)
  (* SCREEN_HEADER                                               *)
  (*    SCREEN HEADER PRINTS A HEADING AT THE TOP OF THE         *)
  (*    TERMINAL SCREEN WHICH INFORMS USER OF THE                *)
  (*    FUNCTION HE IS CURRENTLY IN AS WELL AS INFORMATION       *)
  (*    CONCERNING ANY RELEVANT KEY DATA OF THE RECORD           *)
  (*    TYPE HE IS WORKING ON.                                   *)
  (*                                                             *)
  (*-------------------------------------------------------------*)

  PROCEDURE SCREEN_HEADER (FUN_NUM:INTEGER);

  BEGIN

    CLEAR_SCREEN;
    WRITELN ('            UPDATE MODULE          ');
    PRINTLN_MSG (OUTPUT, MESSAGES, FUN_NUM);
    WRITELN ('*****************************************************');
    WRITELN;
    WRITELN;
    WRITELN    ('SIMULATION MODEL NUMBER:     ', SIMNUM:2);
    IF JOBNUM <> 0 THEN
      WRITELN ('JOB TYPE NUMBER:             ', JOBNUM:2);
    IF RR_S_NUM <> 0 THEN
      WRITELN ('ROUTING REC SERVER NUMBER : ', RR_S_NUM:2);
    WRITELN;
    WRITELN;

  END; (* PROCEDURE  SCREEN HEADER *)
```

```
(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*                                                             *)
(*  PROCEDURE IO-EDIT:                                         *)
(*    PROCEDURE CONTROLS THE DIALOGUE WHICH REQUESTS           *)
(*    USERS TO INPUT VARIOUS DATA FIELDS.  IT ALSO             *)
(*    ERROR CHECKS THE INPUT DATA FOR THOSE FIELDS             *)
(*    AND PROMPTS THE USER FOR RE-INPUT IF THE DATA            *)
(*    VALUE IS IN ERROR.                                       *)
(*                                                             *)
(*-------------------------------------------------------------*)

PROCEDURE IO_EDIT  (MSG_NUM,ED_NUM: INTEGER;
                    VAR INT_INPUT:INTEGER;
                    SUCCESS:BOOLEAN);

VAR  GOOD_INPUT: BOOLEAN;
     EDIT: INTEGER;

BEGIN

    GOOD_INPUT := FALSE;

       REPEAT
         PRINTLN_MSG (OUTPUT, MESSAGES, MSG_NUM);
         READLN (INT_INPUT);

            CASE ED_NUM OF

                 1,2: (* SIMULATION NUMBER *)
                       IF INT_INPUT IN OK_SIMNUM THEN
                         GOOD_INPUT:= TRUE;


                 3:    (* EDIT SG NUMBER *)
                       IF INT_INPUT IN OK_SG_NUM THEN
                          GOOD_INPUT := TRUE;

                 4,5 : (* EDIT ARRIVAL, SERVICE DIST *)
                       IF INT_INPUT IN OK_DIST THEN
                          GOOD_INPUT := TRUE;

                 6,7:  (* EDIT ARRIVAL, SERVICE DIST PARAM *)
```

```
                    IF INT_INPUT  <= MAX_RATE THEN
                       GOOD_INPUT := TRUE;

                 8:    (* EDIT PRIORITY *)
                    IF INT_INPUT IN OK_PRIORITY THEN
                       GOOD_INPUT := TRUE;

                 9:    (* EDIT QUE DISP *)
                    IF INT_INPUT IN OK_QUE_DISP THEN
                       GOOD_INPUT:= TRUE;


                 END; (* CASE EDIT *)

                 IF NOT GOOD_INPUT THEN
                    WRITE ('ERROR IN INPUT. RE');

              UNTIL GOOD_INPUT;

END; (* PROCEDURE IO_EDIT *)
(*-------------------------------------------------------*)
```



```
(*-------------------------------------------------------*)
(*                                                       *)
(*  PROCEDURE ARRAY_IO_EDIT:                             *)
(*    THIS PROCEDURE CONTROLS THE INTERACTIVE DIALOGUE   *)
(*    REQUESTING USER TO INPUT VALUES TO THE ROUTING     *)
(*    PROBABILITY ARRAY IN THE ROUTING RECORD. IT CHECKS *)
(*    THE VALUES AND REQUESTS REINPUT OF THE ARRAY       *)
(*    VALUES IF THE TOTAL OF THE VALUES DOES NOT EQUAL   *)
(*    100.                                               *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE ARRAY_IO_EDIT;

VAR  GOOD_INPUT: BOOLEAN;
     I: INTEGER;
     ROUT_PROB_TOTAL: INTEGER;

BEGIN

        GOOD_INPUT := FALSE;
        REPEAT
```

```
            WRITELN ('ENTER THE ROUTING PROB FROM ',
                     'SERVER GROUP ',RR_S_NUM:2, ' TO:');

            FOR I := 1 TO MAX_SERV DO
               DREC.REC_ARRAY I := 0;

            FOR I := 1 TO MAX_SERV DO
               BEGIN
                 WRITE ('SERVER GROUP ', I:2, ':  ');
                 READLN (DREC.REC_ARRAY I )
               END;

            (* CHECK INPUT *)

            ROUT_PROB_TOTAL := 0;
            FOR I:= 1 TO MAX_SERV DO
              ROUT_PROB_TOTAL := ROUT_PROB_TOTAL +
                                 DREC.REC_ARRAY  I ;
            IF ROUT_PROB_TOTAL = 100 THEN
               GOOD_INPUT := TRUE
            ELSE
               PRINTLN_MSG (OUTPUT, MESSAGES, RR_PROB_ERR)

         UNTIL GOOD_INPUT;

END; (* PROCEDURE *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(*   PROCEDURE S_ARRAY_ID_EDIT                           *)
(*     THIS PROCEDURES CONTROLS INTERACTIVE DIALOGUE AND *)
(*     THE INPUT OF VALUES FOR THE SERVER GROUP RECORD   *)
(*     ARRAY OF NUMBER OF SERVERS IN EACH SERVER GROUP.  *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE S_ARRAY_ID_EDIT;

VAR   I:INTEGER;
      GOOD_INPUT: BOOLEAN;
```

```
BEGIN

  GOOD_INPUT:= FALSE;
  REPEAT
    WRITELN ('ENTER THE NUMBER OF SERVERS FOR');
    FOR I:= 1 TO (MAX_SERV - 1)  DO
      BEGIN
        WRITE ('SERVER GROUP ', I:2, ':  ');
        READLN (DREC.REC_ARRAY[ I ]);
      END;
    GOOD_INPUT := TRUE;
  UNTIL GOOD_INPUT;

END; (* PROCEDURE S_ARRAY_IO_EDIT *)


(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE ADD_REC                                     *)
(*    THIS PROCEDURE ADDS A RECORD TO THE INDEXED        *)
(*    SEQUENTIAL DATABASE.  IT FIRST ATTEMPTS            *)
(*    LOCATE THE RECORD IN THE DATABASE BY RECORD KEY.   *)
(*    IF THE KEY IS NOT FOUND THE RECORD IS ADDED.       *)
(*    IF THE KEY ALREADY EXISTS IN THE DATA BASE         *)
(*    AN ERROR MSG IS DISPLAYED.                         *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE ADD_REC (VAR REC_ADDED: BOOLEAN);

VAR  AR_CHAR: CHAR;

BEGIN
  REC_ADDED:= FALSE;
  WRITELN ('DO WISH TO ADD RECORD TO THE DATA BASE ? Y/-');
  READLN (AR_CHAR);
  IF AR_CHAR = 'Y' THEN
    BEGIN
      FINDK (RECFILE, 0, DREC.RECORD_KEY);
      IF UFB (RECFILE) THEN
        BEGIN
          WRITE (RECFILE, DREC);
```

```
            WRITELN ('RECORD SUCCESSFULLY ADDED ');
            REC_ADDED:= TRUE;
          END
        ELSE
            WRITELN ('RECORD ALREADY EXISTS, NOT ADDED');
      END  (* IF AR CHAR *)

    ELSE
      WRITELN ('RECORD NOT ADDED ');

END; (* PROCEDURE ADD REC *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE PRINT_REC                                   *)
(*   PRINTS THE RECORD FIELDS OF A RECORD TYPE TO        *)
(*   THE SCREEN SO THAT THE USER MAY REVIEW              *)
(*   THE RECORD CONTENTS.                                *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE PRINT_REC (REC_TYPE: INTEGER);

VAR I_CHAR: CHAR;

BEGIN
  CASE REC_TYPE OF
  1: BEGIN

     WRITELN ('ARRIVAL DIST IS:          ',DREC.REC_DIST:2);
     WRITELN ('ARRIVAL DIST PARAM IS:    ', DREC.REC_RATE:2);
     WRITELN ('JOB TYPE PRIORITY IS:     ', DREC.REC_PRIORITY:2);
     WRITELN;

     END;

  2: BEGIN

     WRITELN ('SERVICE DIST IS:           ' ,DREC.REC_DIST:2);
     WRITELN ('SERVICE DIST PARAM IS:     ', DREC.REC_RATE:2);
     FOR I:= 1 TO MAX_SERV DO
```

```
        BEGIN
          WRITE ('ROUTING PROB TO ', I:2);
          WRITELN (' IS:       ', DREC.REC_ARRAY I  :2);
        END; (* FOR *)
      WRITELN;
      END;

  3: BEGIN

     FOR I:= 1 TO MAX_SERV DO
       WRITELN ('NUMBER OF SERVERS IN SG ', I:2,
                ' IS: ', DREC.REC_ARRAY I  :2);
     WRITELN;

     END;

  END; (* CASE REC_TYPE *)


END;   (*PROCEDURE PRINT_REC*)


(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*                                                             *)
(*   PROCEDURE DEL_REC                                         *)
(*     THIS PROCEDURE IS USED TO DELETE RECORDS                *)
(*     FROM THE INDEXED SEQUENTIAL DATABASE.  IF               *)
(*     THE RECORD IS NOT FOUND IN THE DATABASE, AN             *)
(*     ERROR MSG IS SENT TO THE SCREEN SO STATING.             *)
(*                                                             *)
(*-------------------------------------------------------------*)

PROCEDURE DEL_REC;

VAR  INP_CHAR:CHAR;

BEGIN

  FINDK (RECFILE, 0, DREC.RECORD_KEY);
  IF NOT UFB (RECFILE) THEN
    BEGIN
      DREC:= RECFILE↑;
```

```
        DECOMPOSE_KEY;
        PRINT_REC (RECTYPE);
        WRITELN ('DO YOU WISH TO DELETE THIS RECORD? Y/-');
        READLN (INP_CHAR);
        IF INP_CHAR = 'Y' THEN
          BEGIN
            DELETE (RECFILE);
            WRITELN ('RECORD SUCCESSFULLY DELETED');
          END
        ELSE
          WRITELN ('RECORD NOT DELETED');
      END (* IF RECORD EXISTS *)

    ELSE
      WRITELN ('NO RECORD FOUND');

END;  (* PROCEDURE DEL REC *)

(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*                                                             *)
(* PROCEDURE ADD_ROUTING_REC                                   *)
(*   CONTROLS THE SEQUENCE OF EVENTS REQUIRED                  *)
(*   OBTAIN INPUT DATA FROM THE USER, CALCULATE THE            *)
(*   RECORD KEY, AND ADD THE ROUTING RECORD TO THE             *)
(*   DATABASE.                                                 *)
(*                                                             *)
(*-------------------------------------------------------------*)

PROCEDURE ADD_ROUTING_REC (GOOD_JOBNUM: BOOLEAN);

VAR     ARR_CHAR: CHAR;
        ARR_INT: INTEGER;
        ROUT_PROB_TOTAL : INTEGER;
        SERV_GROUP : INTEGER;
        ARR_GOOD_INPUT: BOOLEAN;
        I: INTEGER;
        ED_NUM: INTEGER;
        CORRECT: BOOLEAN;
        PARAM_MSG: INTEGER;
        GOOD_ACD: BOOLEAN;

BEGIN
```

```
ARR_CHAR := ' ';
REPEAT
     SCREEN_HEADER (ADD_RR);
     INITIALIZE_REC;
     IF NOT GOOD_JOBNUM THEN
        BEGIN
          PRINTLN_MSG (OUTPUT, MESSAGES, RR_JOB_NUM);
          READLN (JOBNUM);
          COMPUTE_KEY (JT_REC);
          FINDK (RECFILE, 0, DREC.RECORD_KEY);
          IF NOT UFB (RECFILE) THEN
            GOOD_JOBNUM := TRUE
          ELSE
            WRITELN ('ERROR A JOB TYPE RECORD DOES NOT EXIST');
        END; (* ELSE *)

     IF GOOD_JOBNUM THEN
       BEGIN

     (* ENTER THE SERVER NUMBER *)
         ED_NUM := 3;
         IO_EDIT (RR_SERV_NUM, ED_NUM, RR_S_NUM, CORRECT);

         IF RR_S_NUM <> 0 THEN
           BEGIN

             ED_NUM := 5;
             IO_EDIT (RR_DIST, ED_NUM, DREC.REC_DIST, CORRECT);


             (* GET THE SERVICE PARAM *)
             PARAM_MSG := (RR_DIST * 10 ) + DREC.REC_DIST;
             ED_NUM := 6;
             IO_EDIT (PARAM_MSG, ED_NUM, DREC.REC_RATE, CORRECT);

           END; (* IF RR_S_NUM *)

         ARRAY_IO_EDIT;


      (* ADD THE RECORD TO THE DATA BASE *)

         COMPUTE_KEY (RR_REC);
         SCREEN_HEADER (ADD_RR);
         PRINT_REC (RR_REC);
         ADD_REC (GOOD_ADD);
         RR_S_NUM:= 0;
```

```
            END; (* EIF *)

          WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
          READLN (ARR_CHAR);

      UNTIL (ARR_CHAR = 'Y');

    END; (* PROCEDURE TO ADD ROUTING RECORDS *)


    (*-------------------------------------------------------*)


    (*-------------------------------------------------------*)
    (*                                                       *)
    (* PROCEDURE ADD_JOB_TYPE_REC                            *)
    (*   CONTROLS THE SEQUENCE OF EVENTS REQUIRED TO         *)
    (*   OBTAIN AND EDIT THE JOB TYPE RECORD DATA FROM       *)
    (*   THE USER, COMPUTE THE RECORD KEY, AND ADD           *)
    (*   THE RECORD TO THE DATABASE.                         *)
    (*                                                       *)
    (*-------------------------------------------------------*)

    PROCEDURE  ADD_JOB_TYPE_REC;

    VAR   INT_RESP: INTEGER;
          CHAR_RESP: CHAR;
          CORRECT: BOOLEAN;
          STOP_LOOP: BOOLEAN;
          RECTYPE: INTEGER;
          EDIT_NUM : INTEGER;
          PARAM_MSG: INTEGER;
          GOOD_ADD: BOOLEAN;

    BEGIN (* PROCEDURE ADD JOB_TYPE_REC *)

      INITIALIZE_REC;


      STOP_LOOP := FALSE;
      RECTYPE := 1;
      REPEAT
        JOBNUM := NEXT_JT_NUM;
```

```
RR_S_NUM:= 0;
SCREEN_HEADER (ADD_JT);


EDIT_NUM := 4;
IO_EDIT (JT_DIST, EDIT_NUM, DREC.REC_DIST, CORRECT);

EDIT_NUM := 7;
PARAM_MSG:= JT_DIST*10 + DREC.REC_DIST;
IO_EDIT (PARAM_MSG, EDIT_NUM, DREC.REC_RATE,CORRECT);

EDIT_NUM := 8;
IO_EDIT (JT_PRIORITY, EDIT_NUM, DREC.REC_PRIORITY, CORRECT);

(* COMPUTE THE RECORD KEY *)

COMPUTE_KEY (JT_REC);
(* PRINT THE COMPLETED RECORD TO THE SCREEN *)
SCREEN_HEADER (ADD_JT);
PRINT_REC (JT_REC);

(* ADD THE JOB TYPE RECORD TO THE DATABASE *)
ADD_REC (GOOD_ADD);
(* ONLY GO TO ADD ROUTING REC IF THE JOB TYPE
   RECORD IS SUCCESSFULLY ADDED -- NEED TO PUT
   CHECK IN HERE *)
IF GOOD_ADD THEN
  BEGIN  (* OPTION TO ADD ROUTING RECS *)

    WRITELN ('DO YOU WISH TO ADD ROUTING RECS',
          ' FOR THIS JOB TYPE? Y/-');
    READLN (CHAR_RESP);
    IF CHAR_RESP = 'Y' THEN
      BEGIN (* ADD ROUTING RECS *)
        UM_GOOD_JOBNUM:= TRUE;
        ADD_ROUTING_REC  (UM_GOOD_JOBNUM);
        JOBNUM:= 0;
        RR_S_NUM := 0;
      END; (* ADD ROUTING RECS *)
  END; (* OPTION TO ADD ROUTING RECS *)

(* FIND OUT IF USER WISHES TO STOP OR CONTINUE *)
RR_S_NUM:= 0;
JOBNUM:= 0;
SCREEN_HEADER (ADD_JT);
WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
READLN (CHAR_RESP);
IF (CHAR_RESP = 'Y' ) THEN
```

```
      STOP_LOOP:= TRUE

  UNTIL STOP_LOOP
END; (* PROCEDURE ADD_JOB_TYPE_REC *)
(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
(*                                                       *)
(* PROCEDURE ADD_SERVER_REC                              *)
(*    CONTROLS THE SEQUENCE OF EVENTS REQUIRED TO        *)
(*    OBTAIN AND EDIT THE THE SERVER RECORD              *)
(*    DATA, COMPUTE THE RECORD KEY AND ADD THE           *)
(*    RECORD TO THE DATABASE.                            *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE ADD_SERVER_REC;

VAR INP_CHAR:CHAR;
    GOOD_ADD: BOOLEAN;

BEGIN

  INP_CHAR:= ' ';
  REPEAT
     SCREEN_HEADER (ADD_SERV);
     INITIALIZE_REC;
     S_ARRAY_TO_EDIT;
     SCREEN_HEADER (ADD_SERV);
     PRINT_REC (SG_REC);
     COMPUTE_KEY (SG_REC);
     ADD_REC (GOOD_ADD);
     WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
     READLN (INP_CHAR);
  UNTIL INP_CHAR = 'Y';

END;  (* PROCEDURE ADD SERVER REC *)

(*-------------------------------------------------------*)
```

```
(*-------------------------------------------------------*)
(*                                                       *)
(*  PROCEDURE ENTER_SIM_NUM                              *)
(*    CONTROLS INTERACTIVE DIALOGUE TO REQUEST AND       *)
(*    EDIT A NEW SIMULATION MODEL NUMBER.                *)
(*    ALL RECORD ADDITION AND DELETION REQUESTS ARE      *)
(*    PROCESSED FOR THE SIMULATION NUMBER ENTERED USING  *)
(*    THIS PROCEDURE.                                    *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE ENTER_SIM_NUM;

VAR     SN_IN_OK:  BOOLEAN;

BEGIN  (* ENTER SIM NUM *)

  SN_IN_OK:= FALSE;
  CLEAR_SCREEN;
  REPEAT

    PRINTLN_MSG (OUTPUT, MESSAGES,ENT_SIM_NUM);
    READLN (SIMNUM);
    IF ( SIMNUM >= 0) AND (SIMNUM <= 99) THEN
       SN_IN_OK:= TRUE
    ELSE
       WRITELN ('ERROR IN INPUT, PLEASE REENTER')

  UNTIL SN_IN_OK;

  INITIALIZE_REC;
  COMPUTE_KEY (HD_REC);
  FINDK (RECFILE, 0, DREC.RECORD_KEY);
  IF UFB (RECFILE) THEN
    WRITE (RECFILE,DREC);

END;  (* ENTER SIM NUM *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*                                                       *)
(*  DELETE_JOB_TYPE_RECORDS                              *)
(*    CONTROLS THE DELETION OF JOB TYPE RECORDS          *)
(*    AND ROUTING RECORDS SUBORDINATE TO THE             *)
```

```
(*     JOB TYPES.                                                    *)
(*                                                                   *)
(*-------------------------------------------------------------------*)

PROCEDURE DEL_JOB_TYPE_REC;

VAR         HOLD_KEY: INTEGER;
            EXIT_DEL_JT: BOOLEAN;
            DEL_CHAR: CHAR;


BEGIN   (* PROCEDURE TO DELETE JOB TYPE RECORDS *)

  DEL_CHAR:= ' ';
  EXIT_DEL_JT := FALSE;
  REPEAT

     SCREEN_HEADER (DEL_JT);
     PRINTLN_MSG (OUTPUT, MESSAGES, JT_DEL_NUM);
     READLN (JOBNUM);
     (* CALCULATE THE KEY AND DELETE THE RECORD *)

     COMPUTE_KEY (JT_REC);
     DEL_REC;


  (* NOW DELETE ALL THE ROUTING RECORDS ASSOCIATED WITH THE
     DELETED JOB TYPE RECORD *)

     IF NOT EOF (RECFILE) THEN
     GET (RECFILE);
     DREC:= RECFILE;
     HOLD_KEY := (SIMNUM * 10 ) + RR_REC;
     WHILE ((HOLD_KEY =( TRUNC (DREC.RECORD_KEY / 10000)))
            AND (NOT EOF (RECFILE))) DO
       BEGIN
         DELETE (RECFILE);
         IF NOT EOF (RECFILE) THEN
          GET (RECFILE);
       END;

    WRITELN (' DO YOU WISH TO EXIT FUNCTION? Y/-');
    READLN (DEL_CHAR);
    IF DEL_CHAR = 'Y' THEN
       EXIT_DEL_JT:= TRUE;
  UNTIL EXIT_DEL_JT;
```

```
END; (* PROCEDURE DEL_JT_REC *)

(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
(*                                                       *)
(*   PROCEDURE DEL_ROUTING_REC                           *)
(*     CONTROLS THE DELETION OF ROUTING RECORDS          *)
(*     FROM THE DATA BASE.                               *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE DEL_ROUTING_REC;

VAR DEL_RR_CHAR: CHAR;

BEGIN

  DEL_RR_CHAR:= ' ';
  REPEAT
     SCREEN_HEADER (DEL_RR);
     WRITELN ('ENTER JOB TYPE NUMBER OF ROUTING_REC');
     READLN (JCBNUM);
     WRITELN ('ENTER SERVER GROUP NUMBER');
     READLN (RR_S_NUM);
     COMPUTE_KEY (RR_REC);
     DEL_REC;
     WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
     READLN (DEL_RR_CHAR);
     IF DEL_RR_CHAR <> 'Y' THEN
       BEGIN
         JCBNUM := 0;
         RR_S_NUM:= 0;
       END;
  UNTIL DEL_RR_CHAR = 'Y';

END; (* PROCEDURE DELETE THE ROUTING RECORD *)

(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
```

```
(*                                                                  *)
(* PROCEDURE DEL_SERVER_REC                                         *)
(*   CONTROLS THE DELETION OF SERVER RECORDS FROM                   *)
(*   THE DATABASE.                                                  *)
(*                                                                  *)
(*------------------------------------------------------------------*)

PROCEDURE DEL_SERVER_REC;

VAR DEL_SERV_CHAR:CHAR;

BEGIN

  SCREEN_HEADER (DEL_SERV);
  COMPUTE_KEY (SG_REC);
  DEL_REC;
  WRITELN ('ENTER ANY CHARACTER TO RETURN TO MAIN MENU');
  READLN (DEL_SERV_CHAR);

END; (* PROCEDURE DELETE SERVER RECORD *)

(*------------------------------------------------------------------*)


(*------------------------------------------------------------------*)
(*                                                                  *)
(* PROCEDURE COPY_SIM_MODEL:                                        *)
(*   COPIES ALL THE RECORDS IN THE DATA BASE FOR A                  *)
(*   GIVE SIMULATION NUMBER TO A NEW SIMULATION NUMBER              *)
(*                                                                  *)
(*------------------------------------------------------------------*)

PROCEDURE COPY_SIM_MODEL;

VAR  NEW_SIMNUM: INTEGER;
     OLD_SIMNUM: INTEGER;
     STOP_COPY: BOOLEAN;
     RESP: CHAR;
     OLD_KEY:INTEGER;
     SAVE_SIMNUM: INTEGER;
     REC_COPY: BOOLEAN;

BEGIN
  SAVE_SIMNUM:= SIMNUM;
  RESP := ' ';
```

```
REPEAT
  REC_COPY:= TRUE;
  STOP_COPY:= FALSE;
  CLEAR_SCREEN;
  SIMNUM:= 0;
  JOBNUM:= 0;
  RR_S_NUM:= 0;
  SCREEN_HEADER (COP_SIM);

  WRITELN ('ENTER NUMBER OF MODEL COPYING FROM ');
  READLN (OLD_SIMNUM);
  WRITELN ('ENTER NUMBER OF MODEL COPYING TO ');
  READLN (NEW_SIMNUM);

  (* SEE IF NUMBER COPYING TO ALREADY EXISTS *)

  SIMNUM:= NEW_SIMNUM;
  COMPUTE_KEY (HD_REC);
  FINDK (RECFILE,0, DREC.RECORD_KEY);
  IF NOT UFB (RECFILE) THEN
    BEGIN (* IF MODEL NOT IF DB *)
      STOP_COPY:= TRUE;
      REC_COPY:= FALSE;
      WRITELN ('SIMULATION MODEL NUMBER ', SIMNUM:3,
               ' ALREADY EXISTS ON DATA BASE');
    END; (* IF MODEL NOT IN DB *)

  (* SEE IF NUMBER COPYING FROM EXISTS *)

  SIMNUM:= OLD_SIMNUM;
  COMPUTE_KEY (HD_REC);
  FINDK (RECFILE, 0,DREC.RECORD_KEY);
  IF UFB (RECFILE) THEN
    BEGIN
      REC_COPY:= FALSE;
      STOP_COPY:= TRUE;
      WRITELN ('SIMULATION MODEL NUMBER ', SIMNUM:3,
               ' DOES NOT EXIST ON DATA BASE');
    END;

  WHILE NOT STOP_COPY DO
    BEGIN (* WHILE NOT STOP_COPY *)
      DREC:= RECFILE^;
      OLD_KEY:= DREC.RECORD_KEY;
      DECOMPOSE_KEY;
      IF SIMNUM <> OLD_SIMNUM THEN
        STOP_COPY := TRUE
      ELSE
```

```
            BEGIN (* COPY *)
              SIMNUM:= NEW_SIMNUM;
              COMPUTE_KEY(RECTYPE);
              FINDK (RECFILE, 0, DREC.RECORD_KEY);
              WRITE (RECFILE, DREC);
              DREC.RECORD_KEY:= OLD_KEY;
              FINDK (RECFILE, 0, DREC.RECORD_KEY);
              IF NOT EOF (RECFILE) THEN
                GET (RECFILE )
              ELSE (* STOP BECAUSE ITS EOF *)
                STOP_COPY := TRUE;
            END; (* COPY *)
        END; (* WHILE NOT STOP COPY *)

      IF REC_COPY THEN
        WRITELN ('SIMULATION MODEL COPIED');
      WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
      READLN (RESP);
    UNTIL( RESP = 'Y');
    SIMNUM:= SAVE_SIMNUM;

  END; (* PROCEDURE COPY SIMULATION MODEL *)


  (*-------------------------------------------------------*)


  (*-------------------------------------------------------*)
  (*                                                       *)
  (* PROCEDURE DEL_SIM_MODEL:                              *)
  (*   DELETES ALL THE RECORDS IN THE DATA BASE FOR A      *)
  (*   GIVEN SIMULATION MODEL NUMBER                       *)
  (*                                                       *)
  (*-------------------------------------------------------*)

  PROCEDURE DEL_SIM_MODEL;

  VAR  RESPCHAR: CHAR;
       SAVE_SIMNUM:INTEGER;
       STOP_DELETE: BOOLEAN;

  BEGIN

      RESPCHAR:= ' ';
      STOP_DELETE := FALSE;
      CLEAR_SCREEN;
      SCREEN_HEADER (DEL_SIM);
      WRITELN ('ENTER NUMBER OF SIMULATION MODEL TO DELETE');
```

```
        READLN (SIMNUM);
        SAVE_SIMNUM:= SIMNUM;
        COMPUTE_KEY (HD_REC);
        FINDK (RECFILE, 0, DREC.RECORD_KEY);
        IF UFB (RECFILE) THEN
          WRITELN ('SIMULATION MODEL DOES NOT EXIST')
        ELSE
          BEGIN
            WRITELN ('DO YOU WISH TO DELETE SIMULATION MODEL ',
                     SIMNUM:2, '? Y/-');
            READLN (RESPCHAR);
          END;
      IF RESPCHAR = 'Y' THEN
          BEGIN
            WHILE NOT STOP_DELETE DO
              BEGIN
                DREC:= RECFILE^;
                DECOMPOSE_KEY;
                IF SAVE_SIMNUM <> SIMNUM THEN
                  STOP_DELETE:= TRUE
                ELSE
                  BEGIN
                    DELETE (RECFILE);
                    GET (RECFILE);
                  END;
              END; (* WHILE NOT STOP DELETE *)
          END; (* IF *)
        WRITELN ('DO YOU WISH TO EXIT FUNCTION? Y/-');
        READLN (RESPCHAR);

      UNTIL (RESPCHAR = 'Y');

    END; (* PROCEDURE DELETE SIM MODEL *)

    (*-------------------------------------------------------------*)
    (*-------------------------------------------------------------*)


    BEGIN  (*PROCEDURE UPDATE_MENU *)

      RESETK (RECFILE, 0);

      (* MAKE ASSIGNMENTS TO SETS FOR ERROR CHECKING PURPOSES *)

      OK_SIMNUM:=  0..MAX_SIM ;
      OK_SG_NUM:=  0..MAX_SERV ;
      OK_DIST:=  0..MAX_DIST ;
```

```
    OK_PRIORITY:=  0..MAX_PRIORITY ;
    OK_QUE_DISP:=  1..MAX_QUE_DISP ;

    ENTER_SIM_NUM;
    UM_STOP:= FALSE;

    REPEAT

      UM_INPUT_ERR := FALSE;
      REPEAT
        JOBNUM:= 0;
        RR_S_NUM:= 0;
        SCREEN_HEADER (MAIN_MEN);
        PRINTLN_MSG (OUTPUT, MESSAGES, UPD_MENU);
        READLN (UPD_OPT);
        IF (UPD_OPT <= 11) AND (UPD_OPT >= 0) THEN
           UM_INPUT_ERR:= TRUE
        ELSE
           WRITELN ('ERROR IN INPUT TRY AGAIN ')
      UNTIL UM_INPUT_ERR;

      CASE UPD_OPT OF

       1: ENTER_SIM_NUM;
       2: ADD_JOB_TYPE_REC;
       3: BEGIN
            UM_GOOD_JOBNUM:= FALSE;
            ADD_ROUTING_REC (UM_GOOD_JOBNUM);
          END;
       4: ADD_SERVER_REC;
       5: DEL_JOB_TYPE_REC;
       6: DEL_ROUTING_REC;
       7: DEL_SERVER_REC;
       9: COPY_SIM_MODEL;
      10: DEL_SIM_MODEL ;
      11: UM_STOP:= TRUE

      END  (*CASE UPD_OPT *)

    UNTIL UM_STOP;

  END; (* PROCEDURE UPDATE *)

  (*-------------------------------------------------*)
  (*                END UPDATE MODULE                *)
  (*-------------------------------------------------*)
```

```
{*-------------------------------------------------------*)
{*-------------------------------------------------------*)
{*          CHECK SIM SPECS MODULE                       *)
{*-------------------------------------------------------*)
{*-------------------------------------------------------*)


{*-------------------------------------------------------*)
{* CHECK_SIM_SPECS PROCEDURE                             *)
{*                                                       *)
{* CALLED BY:  MAIN CPMT DRIVER                          *)
{* CALLS: ROUTING_REC_CHECK                              *)
{*                                                       *)
{* THIS PROCEDURE CHECKS THE DATA BASE RECORDS FOR A     *)
{* GIVEN SIMULATION MODEL NUMBER TO DETERMINE WHETHER    *)
{* THE SIM MODEL SPECIFICATIONS ARE COMPLETE AND CAN     *)
{* BE EXECUTED BY THE EXT MODULE.                        *)
{*                                                       *)
{*-------------------------------------------------------*)

PROCEDURE CHECK_SIM_SPECS (CHECK_SIMNUM :INTEGER;
                           VAR     CHECK: BOOLEAN);

VAR  LIMIT : INTEGER;
     CHECK_ARRAY: ARRAY  0..MAX_SERV  OF CHAR;
     STOP_PROCEDURE: BOOLEAN;
     FIRST_JOBTYPE: BOOLEAN;
     LAST_JOBNUM: INTEGER;
     SERVER_REC: BOOLEAN;

{*-------------------------------------------------------*)
{* PROCEDURE ROUTING_REC_CHECK                           *)
{*                                                       *)
{* CALLED BY: CHECK_SIM_SPECS                            *)
{*                                                       *)
{* THIS PROCEDURE IS CALLED EVERY TIME A JOB TYPE REC    *)
{* IS ENCOUNTERED (EXCEPT FOR THE FIRST JOB REC)         *)
{* IN ORDER TO EVALUATE THE ROUTING RECORD SPECS FOR     *)
{* THE PREVIOUS JOB TYPE.  ALSO CALLED AT THE END        *)
{* TO EVALUATE ROUTING REC SPECS OF LAST JOB TYPE.       *)
{* THE PROCEDURE LOOKS FOR CERTAIN CONDITIONS IN THE     *)
{* CHECK_ARRAY.  THE CHECK_ARRAY IS 'FILLED OUT' AS THE  *)
{* ROUTING RECS ASSOCIATED WITH A GIVEN JOB TYPE ARE     *)
{* PROCESSED BY THE CHECK_SIM_SPECS PROCEDURE.           *)
{*-------------------------------------------------------*)
```

```
PROCEDURE ROUTING_REC_CHECK;
VAR NO_RR:BOOLEAN;
BEGIN
  (* CHECK IF NO ROUTING RECORDS FOR JOBTYPE.
     IF THE ARRAY IS ALL BLANKS THERE ARE NO ROUTING RECS *)
  NO_RR:= TRUE;
  FOR I:= 1 TO MAX_SERV DO
    IF CHECK_ARRAY  I  <> ' ' THEN
      NO_RR := FALSE;
  IF NO_RR THEN
    BEGIN
      WRITELN (OUTFILE, 'NO ROUTING RECORDS FOR JOB TYPE ',
               LAST_JOBNUM:2);
      CHECK:= FALSE;
    END; (* IF NO RR *)

  (* CHECK TO SEE IF THERE IS A ROUTING RECORD FOR SERVER
     GROUP 0.  THERE SHOULD BE AN F IN THE ARRAY ITEM
     INDEXED BY SG 0 *)

  IF CHECK_ARRAY  0  <> 'F' THEN
    BEGIN (* NO SG 0 RR *)
      WRITELN (OUTFILE, 'NO SERVER GROUP 0 ROUTING RECORD ',
               'FOR JOB TYPE ', LAST_JOBNUM:2);
      CHECK:= FALSE;
    END; (* NO SG 0 RR *)

  (* CHECK IF JOB TYPE NOT ROUTED TO EXIT. THERE SHOULD
     BE A 'T' IN THE ARRAY ITEM INDEXED BY THE EXIT SERVER
     GROUP NUMBER TO INDICATE THAT THE JOB IS ROUTED TO
     THE EXIT *)

  IF CHECK_ARRAY  MAX_SERV  <> 'T' THEN
    BEGIN
      WRITELN (OUTFILE, 'JOB TYPE ', LAST_JOBNUM:2,
               ' NOT ROUTED TO EXIT SERVER GROUP');
      CHECK:= FALSE
    END;

  (* CHECK IF ROUTED TO A SERVER GROUP BUT NOT
     FROM THE SERVER GROUP.  IF ANY SERVER GROUP
     OTHER THAT EXIT SERVER GROUP HAS A 'T' VALUE THAT
     MEANS THE JOB HAS BEEN ROUTED TO THE SERVER GROUP
```

```
      WITHOUT BEING ROUTED FROM IT *)

    FOR I:= 1 TO LIMIT DO
      IF CHECK_ARRAY  I  = 'T' THEN
        BEGIN
          WRITELN (OUTFILE, 'JOB TYPE ', LAST_JOBNUM:2,
                   ' ROUTED TO BUT NOT FROM SG ',I:2);
          CHECK := FALSE;
        END;

    END; (* PROCEDURE ROUTING REC CHECK *)

  (*-------------------------------------------------------*)
  (*-------------------------------------------------------*)

  BEGIN  (* PROCEDURE CHECK SIM SPECKS *)

    CHECK:= TRUE;
    LAST_JOBNUM:= 0;
    FIRST_JOBTYPE:= TRUE;
    SERVER_REC:= FALSE;
    SIMNUM:= CHECK_SIMNUM;

    (* CHECK FOR ERROR #1: EXISTENCE OF SIMULATION NUMBER IN DB *)

    COMPUTE_KEY (HD_REC);
    FINDK (RECFILE, 0, DREC.RECORD_KEY);
    IF UFB (RECFILE) THEN
      BEGIN
        WRITELN (OUTFILE, 'SIMULATION NUMBER DOES NOT EXIST ');
        CHECK:= FALSE;
      END
    ELSE (* SIMULATION NUMB EXISTS *)
      BEGIN
        IF NOT EOF (RECFILE) THEN
          GET (RECFILE);
        DREC:= RECFILE^;
        DECOMPOSE_KEY;
        LIMIT := MAX_SERV - 1;

        WHILE (SIMNUM = CHECK_SIMNUM) AND NOT EOF (RECFILE) DO
          BEGIN
          CASE RECTYPE OF
```

```
1:   BEGIN (* JOB TYPE RECORD *)
       IF NOT FIRST_JOBTYPE THEN
          ROUTING_REC_CHECK;

       (* CHECK TO SEE IF JOB TYPE NUMBERS ARE SEQUENTIAL *)

       FIRST_JOBTYPE:= FALSE;
       IF JOBNUM <> LAST_JOBNUM + 1 THEN
         BEGIN
           WRITELN (OUTFILE, 'JOB NUMBERS ARE NOT SEQUENTIAL ');
           CHECK:= FALSE;
         END;
       LAST_JOBNUM := JOBNUM;

       (* INITIALIZE THE CHECK_ARRAY WHICH WILL BE USED TO
          CHECK THE ROUTING RECORDS FOR THE JOB TYPE RECORD *)

       FOR I := 0 TO MAX_SERV DO
         CHECK_ARRAY  I   := ' ';
     END; (* IF JOBTYPE RECORD *)


2:   BEGIN

       (* PUT AN 'F' IN THE ARRAY ITEM INDEXED BY
          THE SERVER GROUP NUMBER OF THE ROUTING RECORD
          TO INDICATE THAT THE JOB TYPE HAS ROUTING
          PROBABILITY SPECIFIED FROM THAT SERVER GROUP *)

       CHECK_ARRAY  RR_S_NUM := 'F';

       (* NOW LOOK AT THE ROUTING PROBABILITY ARRAY AND
          FOR ALL THE SERVER GROUP DESTINATION
          POSSIBILITIES PUT A 'T' IN THE ARRAY INDEXED BY
          THE SG DESTINATION IF THERE IS NOT ALREADY AN 'F'
          THERE.  THIS INDICATES THE JOB TYPE IS
          ROUTED TO THE SERVER GROUP BUT HAS NOT YET
          BEEN ROUTED FROM THE SERVER GROUP  *)


       FOR I := 1 TO MAX_SERV DO
         BEGIN  (* FOR *)
           IF DREC.REC_ARRAY  I  <> 0  THEN
             IF CHECK_ARRAY  I  <> 'F' THEN
               CHECK_ARRAY  I  := 'T';
         END;  (* FOR *)

       IF RR_S_NUM <> 0 THEN
```

```
                    (* IF THE ROUTING PROB SPECIFIES 100% ROUTING
                       PROB BACK TO THE SAME SERVER GROUP, THEN
                       THERE IS A ROUTING LOOP ERROR *)

                    IF DREC.REC_ARRAY  RR_S_NUM = 100 THEN
                      BEGIN
                        WRITELN (OUTFILE, 'JOB TYPE ',
                                 JOBNUM:2, ' SERVER GROUP ',
                                 RR_S_NUM:2, ' ROUTING LOOP');
                        CHECK := FALSE;
                      END;
                END; (* IF RECTYPE = 2 *)
            3:  SERVER_REC:= TRUE

            END; (* CASE RECTYPE *)

               IF NOT EOF (RECFILE) THEN
                 BEGIN
                   GET (RECFILE);
                   DREC:= RECFILE¬;
                   DECOMPOSE_KEY;
                 END; (* IF NOT EOF *)
          END; (* WHILE *)
        END; (* SIMULATION NUMB EXISTS *)

      ROUTING_REC_CHECK;
      IF NOT SERVER_REC THEN
        WRITELN (OUTFILE, 'SERVER RECORD DOES NOT EXIST');

      END; (* PROCEDURE CHECK SIM SPECS*)

(*-------------------------------------------------------------*)
(*        END CHECK SIM SPECS MODULE                           *)
(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)
(*              CREATE JOB STREAM MODULE                       *)
(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)


(*-------------------------------------------------------------*)
(* PROCEDURE CREATE_JOB_STREAM  (CJS)                          *)
```

```
(*                                                                    *)
(* THIS PROCEDURE CREATES THE LINKED LIST OF JOB AND                  *)
(* EVENT RECORDS WHICH BECOMES THE ARRIVAL QUEUE FOR                  *)
(* THE EXECUTE_AND_TABULATE PROCEDURE.  IT ACCESSES THE               *)
(* SIMULATION MODEL DATA BASE AND UPLOADS THE MODEL                   *)
(* SPECIFICATION DATA INTO AN INTERMEDIARY LINKED LIST                *)
(* OF JOB TYPE AND ROUTING RECORDS FROM WHICH THE                     *)
(* CREATE_JOB PROCEDURE ACCESSES THE DATA TO GENERATE                 *)
(* JOB RECORDS.                                                       *)
(*  *)
(**)
(*     PARAMETERS:                                                    *)
(*                                                                    *)
(*                  OUTPUT PARAMETER:  ARRIVEQPTR POINTS              *)
(*                  TO THE FIRST JOB RECORD IN THE LINKED             *)
(*                  LIST OF JOB RECORDS AND SUBORDINATE               *)
(*                  EVENT RECORDS WHICH IS THE JOB STREAM             *)
(*                  CREATED BY THIS PROCEDURE                         *)
(**)
(*     PROCEDURES/FUNCTIONS:  MORE DETAILED DESCRIPTION               *)
(*         OF THE PROCEDURES AND FUNCTIONS CALLED BY                  *)
(*         CREATE JOB STREAM PROCEDURE ARE INCLUDED IN                *)
(*         THEIR DECLARATIONS.                                        *)
(**)
(*         PROCEDURE                CALLS                             *)
(*         ------------------------------------                       *)
(*                                                                    *)
(*         CREATE_JOB_STREAM        BUILD_LL_FROM_DB                  *)
(*                                  CREATE_JOB                        *)
(*                                  INSERT_IN_QUEUE                   *)
(*                                                                    *)
(*         BUILD_LL_FROM_DB         PROCESS_JOB_TYPE_DATA             *)
(*                                  PROCESS_ROUTING_DATA              *)
(*                                  PROCESS_SERVER_DATA               *)
(*                                                                    *)
(*         CREATE_JOB               GENERATE_VAL                      *)
(*                                  DESTINATION                       *)
(*                                                                    *)
(*         GENERATE_VAL             MTH$RANDOM                        *)
(*                                                                    *)
(*         INSERT_IN_QUEUE               --                           *)
(*                                                                    *)
(*         DESTINTAION                   --                           *)
(*                                                                    *)
(*                                                                    *)
(*         MTH$ RANDOM                   --                           *)
(*                                                                    *)
(*--------------------------------------------------------------------*)
```

```
PROCEDURE CREATE_JOB_STREAM
    (VAR ARRIVEQPTR:PTR; TOTALJOBS:INTEGER) ;

VAR     HOLD, NEWJOBPTR, ENDQPTR: PTR;

        TEMP_JTPTR: PTR;
        JOBTPTR: PTR;  (* POINTS TO FIRST REC IN LL OF
                          JOB TYPE/ROUTING RECORDS *)
        FIRST_JOB: BOOLEAN;
        JOBCCNT : INTEGER;

(*-------------------------------------------------------------*)

 EXTERNAL, ASYNCHRONOUS  FUNCTION MTH$RANDOM
    (VAR SEED: INTEGER) : REAL; EXTERN;

(*-------------------------------------------------------------*)
(* PROCEDURE BUILD_LL_FROM_DB                                  *)
(*                                                             *)
(* THIS PROCEDURE UPLOADS THE SIMULATION MODEL RECORDS         *)
(* FROM THE DATA BASE INTO THE FORMAT IN WHICH THE DATA        *)
(* IS ACCESSED BY THE CREATE_JOB_STREAM AND EXECUTE_           *)
(* AND_TABULATE MODULES TO CREATE THE JOBS AND SERVER          *)
(* GROUPS TO RUN THE SIMULATION.                               *)
(*                                                             *)
(* IT BUILDS THE JOB TYPE/ROUTING RECORD LINKED LIST           *)
(* FROM THE DATA BASE JOB TYPE AND ROUTING RECORDS. IT         *)
(* READS THE DATA BASE SERVER GROUP RECORD DATA INTO           *)
(* THE NUM_SERVERS ARRAY.  THE JOB TYPE/ROUTING RECORD         *)
(* LINKED LIST IS USED BY THE CREATE_JOB PROCEDURE.            *)
(* THE NUM_SERVERS ARRAY IS USED BY THE CREATE_SERVER_         *)
(* GROUP PROCEDURE IN THE EXECUTE AND TABULATE MODULE.         *)
(*                                                             *)
(*-------------------------------------------------------------*)


PROCEDURE BUILD_LL_FROM_DB (VAR JOB_TYPTR: PTR );

VAR   LASTE, LASTJ : PTR;
      HOLD_SIMNUM : INTEGER;

(*-------------------------------------------------------------*)
```

```
(*-------------------------------------------------------*)
(* PROCEDURE PROCESS_JOB_TYPE_DATA                       *)
(*                                                       *)
(* CALLED BY:  BUILD_LL_FROM_DB                          *)
(*                                                       *)
(* CREATES THE DYNAMIC RECORD JOBS AND READS THE DATA    *)
(* FROM THE DB JOB TYPE RECORD INTO THE JOBS RECORD.     *)
(* LINKS THE NEW JOBS RECORD TO THE PREVIOUS JOBS        *)
(* RECORD VIA THE NEXTJ POINTER FIELD OF THE JOBS        *)
(* RECORD OR ESTABLISHES JOBS RECORD AS THE FIRST        *)
(* RECORD IN THE JOB TYPE/ROUTING LINKED LIST POINTED    *)
(* TO BY THE JOB_TYPTR.                                  *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE PROCESS_JOB_TYPE_DATA;

VAR JT_TEMP :PTR;

BEGIN

  IF JOB_TYPTR = NIL THEN
    LASTJ:= NIL;
  NEW (JT_TEMP, JOBS);
  JT_TEMP@.TAG := JOBS;
  IF LASTJ <> NIL THEN
    LASTJ@.NEXTJ := JT_TEMP
  ELSE
    JOB_TYPTR := JT_TEMP;
  IF LASTJ <> NIL THEN
    JT_TEMP@.JOB_TYPE_JT := LASTJ@.JOB_TYPE_JT + 1
  ELSE
    JT_TEMP@.JOB_TYPE_JT := 1;
  JT_TEMP@.ARRIVAL_RATE := DREC.REC_RATE;
  JT_TEMP@.ARRIVAL_DIST := DREC.REC_DIST;
  JT_TEMP@.PRIORITY_JT := DREC.REC_PRIORITY;
  JT_TEMP@.NEXTJ := NIL;
  LASTJ := JT_TEMP;
  LASTE := NIL

END;

(*-------------------------------------------------------*)
```

```
(*-------------------------------------------------------*)
(* PROCEDURE PROCESS_ROUTING_DATA                        *)
(*                                                       *)
(* CALLED BY: BUILD_LL_FROM_DB                           *)
(*                                                       *)
(* CREATES THE DYNAMIC ROUTING RECORD AND READS THE      *)
(* DATA FROM THE DB ROUTING RECORD INTO IT.  IF THE      *)
(* ROUTING RECORD IS THE FIRST ROUTING RECORD FOR THE    *)
(* JOB TYPE IT LINKS  THE ROUTING RECORD TO THE JOBS     *)
(* RECORD VIA THE JOBS ROUTINGJ FIELD POINTER.  OTHER-   *)
(* WISE IT LINKS THE ROUTING RECORD TO THE LAST ROUTING  *)
(* RECORD VIA THE ROUTING RECORD NEXTR POINTER FIELD     *)
(*                                                       *)
(*-------------------------------------------------------*)



PROCEDURE PROCESS_ROUTING_DATA;

VAR N,I :INTEGER;
    RR_TEMP: PTR;

 BEGIN

  NEW (RR_TEMP, ROUTING);

  IF LASTE = NIL THEN
    LASTJ@.ROUTINGJ := RR_TEMP
  ELSE
    LASTE@.NEXTR := RR_TEMP;
  RR_TEMP@.NEXTR := NIL;


  RR_TEMP@.SERVER_GROUP_R := RR_S_NUM;
  RR_TEMP@.SERVICE_DIST := DREC.REC_DIST;
  RR_TEMP@.SERVICE_RATE := DREC.REC_RATE;
  FOR I:= 1 TO 10 DO
       RR_TEMP@.ROUTING_PROB  I  := DREC.REC_ARRAY  I ;
  LASTE:= RR_TEMP
END;

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
```

```
(* PROCEDURE_PROCESS_SERVER_DATA                                    *)
(*                                                                  *)
(* CALLED BY: BUILD_LL_FROM_DB                                      *)
(*                                                                  *)
(* READS THE DATA FROM THE DATA BASE SERVER GROUP                   *)
(* RECORD INTO THE ARRAY NUM_SERVERS.                               *)
(*                                                                  *)
(*------------------------------------------------------------------*)

PROCEDURE PROCESS_SERVER_DATA;

VAR I:INTEGER;

BEGIN (* PROCESS SERVER DATA *)

    FOR I:= 1 TO MAX_SERV DO
    NUM_SERVERS  I  := DREC.REC_ARRAY  I ;

END;  (* PROCESS SERVER DATA *)

(*------------------------------------------------------------------*)

BEGIN  (* PROCEDURE BUILD_LL_FROM_DB *)

  JOB_TYPTR := NIL;
  HOLD_SIMNUM := SIMNUM;
  COMPUTE_KEY (HD_REC);
  FINDK (RECFILE, 0, DREC.RECORD_KEY);
  DREC:= RECFILE↑;
  DECOMPOSE_KEY;
  WHILE (SIMNUM = HOLD_SIMNUM) AND NOT EOF (RECFILE)  DO
    BEGIN  (* WHILE SIMNUM *)
      CASE RECTYPE OF

        1:  PROCESS_JOB_TYPE_DATA;

        2:  PROCESS_ROUTING_DATA;

        3:  PROCESS_SERVER_DATA;

      END; (* CASE RECTYPE OF *)

      IF NOT EOF (RECFILE) THEN
        BEGIN
          GET (RECFILE);
          DREC:= RECFILE↑;
```

```
          DECOMPOSE_KEY;
        END;
    END; (* WHILE SIMNUM *)
END; (* PROCEDURE BUILD_LL_FROM_DB *)

(*-------------------------------------------------------*)

(*-------------------------------------------------------*)
(* FUNCTION  GENERATE_VAL.                               *)
(*                                                       *)
(* CALLS :  MTH$RANDOM                                   *)
(*                                                       *)
(* CALLED BY: CREATE_JOB                                 *)
(*                                                       *)
(* THIS FUNCTION GENERATES A RANDOM VARIATE BASED        *)
(* UPON DISTRIBUTION TYPE AND A DISTRIBUTION PARAMETER   *)
(* WHICH IS PASSED TO IT.   THE FUNCTION RETURNS AN      *)
(* INTEGER VALUE.  MINIMUM VALUE IS '1'.  A '0' VALUE    *)
(* IS NOT RETURNED IN ORDER TO AVOID SITUTATIONS WHERE   *)
(* PROCESSING TIME OR INTERARRIVAL TIMES ARE '0'.        *)
(* THE RANDOM VARIATE PRODUCED BY THIS FUNCTION IS       *)
(* USED TO DETERMINE JOB INTERARRIVAL TIMES AND THE      *)
(* PROCESSING TIMES OF JOB EVENTS.                       *)
(*                                                       *)
(*-------------------------------------------------------*)

FUNCTION GENERATE_VAL (DIST_TYPE: INTEGER;
                       RATE: INTEGER ) : INTEGER;

VAR   RANDOM_NUMBER : REAL;

    BEGIN  (* GENERATE_VAL *)

      RANDOM_NUMBER:= MTH$RANDOM (SEED);
      CASE DIST_TYPE OF

          1 :  (* DETERMINATE DISTRIBUTION  *)

               GENERATE_VAL := RATE;

          2 :  (* POISSON DISTRIBUTION *)

               GENERATE_VAL :=
                  TRUNC( - RATE * LN ( RANDOM_NUMBER)) + 1);
```

```
            3 :  (* UNIFORM DISTRIBUTION *)
                 GENERATE_VAL :=( TRUNC (RATE * RANDOM_NUMBER) + 1);
        END (* CASE DIST_TYPE *)
    END; (* GENERATE VAL PRODEDURE *)

(*-------------------------------------------------------*)
(* DESTINATION FUNCTION.                                 *)
(*                                                       *)
(* CALLED BY:  CREATE_JOB PROCEDURE.                     *)
(*                                                       *)
(* DESTINATION FUNCTION RETURNS THE SERVER GROUP NUM     *)
(* AT WHICH THE NEXT JOB EVENT WILL BE PROCESSED.        *)
(* THE POINTER ROUTINGPTR POINTS TO THE ROUTING RECORD   *)
(* OF THE SERVER GROUP AT WHICH THE LAST JOB EVENT       *)
(* WAS PROCESSED.  THE ROUTING_PROB ARRAY OF THAT        *)
(* ROUTING RECORD CONTAINS THE PROBABILITIES THAT        *)
(* THE JOB WILL BE ROUTED FROM THE CURRENT SERVER GROUP  *)
(* TO THE OTHER SERVER GROUPS IN THE MODEL.  A RANDOM    *)
(* NUMBER (INTEGER VALUE FROM 0 TO 99) IS MAPPED         *)
(* AGAINST THE PROBABILITIES IN THE ARRAY.               *)
(* IF THE RANDOM INTEGER FALLS WITHIN THE PROBABILITY    *)
(* RANGE FOR A SERVER GROUP, THAT SERVER GROUP BECOMES   *)
(* THE NEXT JOB DESTINATION.                             *)
(*-------------------------------------------------------*)

FUNCTION  DESTINATION (ROUTINGPTR : PTR)      : INTEGER;

VAR       RAND_INT, UPBOUND, LOWBOUND, I, PROB : INTEGER;
          MAP_RANGE:  BOOLEAN;


BEGIN

          RAND_INT:= TRUNC (MTH$RANDOM (SEED) * 100 );
          UPBOUND:= 0;
          LOWBOUND:= 0;
          MAP_RANGE := FALSE;
          I:= 0;


          REPEAT
            I  := I + 1;
            PROB:= ROUTINGPTR@.ROUTING_PROB(.I.);
```

```
            IF PROB <> 0 THEN
               BEGIN
                 UPBOUND:= LOWBOUND + PROB - 1;
                 IF (RAND_INT >= LOWBOUND) AND
                    (RAND_INT <= UPBOUND ) THEN
                      MAP_RANGE:= TRUE
                 ELSE
                    LOWBOUND:= UPBOUND + 1
               END;

          UNTIL MAP_RANGE;

          DESTINATION:= I

END;  (*FUNCTION DESTINATION *)

(*-------------------------------------------------*)

(*-------------------------------------------------*)
(* PROCEDURE CREATE_JOB                            *)
(*                                                 *)
(* CALLS:  GENERATE_VAL, DESTINATION               *)
(*                                                 *)
(* CALLED BY: CREATE_JOB-STREAM                    *)
(*                                                 *)
(* CREATE JOB CREATES A SINGLE JOB RECORD AND ITS  *)
(* ASSOCIATED EVENTS RECORDS.  THE JOB CREATED IS OF *)
(* THE JOB TYPE (JOBS      RECORD) POINTED TO BY THE *)
(* JOBTPTR VARIABLE PARAMETER.  THE NEWLY CREATED  *)
(* JOB IS POINTED TO BY THE POINTER NEWJOBPTR.     *)
(*                                                 *)
(*-------------------------------------------------*)

PROCEDURE CREATE_JOB
                        (JOBTPTR:PTR;
                         VAR NEWJOBPTR:PTR);


VAR     SERV_GROUP: INTEGER;
        ROUTINGPTR: PTR;
        NEWEVENTPTR, LASTEVENTPTR : PTR;
        EVENTPTR: PTR;
        TOT_TIME: INTEGER;
        EVENT_NUM: INTEGER;

   BEGIN (* PROCEDURE CREATE_JOB *)

      (* CREATE THE JOB RECORD *)
```

```
NEW (NEWJOBPTR, JOB_RECORD);
NEWJOBPTR@.TAG:= JOB_RECORD;

(* THE JOB RECORD JOB TYPE AND PRIORITY FIELDS
   CORRESPOND TO FIELDS IN THE JOB TYPE RECORD.
   THE ARRIVAL TIME IS CALCULATED FROM THE
   ARRIVAL DISTRIBUTION AND DISRIBUTION PARAMETER.
   ACTUALLY THE VALUE IN THE ARRIVAL_TIME FIELD
   AS CALCULATED IN THIS PROCEDURE IS
   THE INTER-ARRIVAL TIME OF THE JOB.  THE
   ARRIVAL TIME WILL BE CALCULATED OUTSIDE OF THIS
   PROCEDURE AS THE ARRIVAL_TIME OF THE THE LAST
   JOB OF THIS JOB TYPE PLUS THE INTER-ARRIVAL TIME
   VALUE *)

NEWJOBPTR@.JOB_TYPE:= JOBTPTR@.JOB_TYPE_JT;
NEWJOBPTR@.PRIORITY:= JOBTPTR@.PRIORITY_JT;
NEWJOBPTR@.ARRIVAL_TIME:= GENERATE_VAL (JOBTPTR@.ARRIVAL_DIST,
                                        JOBTPTR@.ARRIVAL_RATE);
NEWJOBPTR@.NEXT_JOB:= NIL;

NEWEVENTPTR := NIL;
LASTEVENTPTR := NIL;
ROUTINGPTR:= JOBTPTR@.ROUTINGJ;
SERV_GROUP := DESTINATION (ROUTINGPTR);
EVENT_NUM := 0;

(*  THE ASSOCIATED JOB EVENTS RECORDS ARE CREATED FOR THE
    JOB RECORD.  THE FOLLOWING LOOP CREATES JOB EVENTS UNTIL
    THE JOB IS ROUTED TO THE EXIT SERVER GROUP (SERVER GROUP
    10).  *)

REPEAT

  EVENT_NUM := EVENT_NUM + 1;
  LASTEVENTPTR := NEWEVENTPTR;

  (* CREATE THE EVENT RECORD *)

  NEW (NEWEVENTPTR, EVENT_RECORD);
  NEWEVENTPTR@.TAG:= EVENT_RECORD;

  (* THE EVENT SERVER GROUP NUMBER IS
     DETERMINED FROM THE ROUTING PROBABILITY
     ARRAY OF THE ROUTING RECORD FOR THE
     SERVER GROUP OF THE LAST EVENT, OR
     FROM THE ROUTING PROBABILITY ARRAY FOR
```

```
      SG 0 IF IT IS THE FIRST EVENT *)

    NEWEVENTPTR@.SERVER_GROUP_NO := SERV_GROUP;
    NEWEVENTPTR@.JOB_PART_NO := EVENT_NUM;
    NEWEVENTPTR@.NEXT_JOB_PART := NIL;

    (* FIND THE ROUTING RECORD
       FOR THE SERVER GROUP OF THE EVENT IN ORDER
       TO ACCESS DATA REQUIRED TO DETERMINE
       EVENT PROCESSING TIME AND SERVER GROUP
       OF NEXT EVENT *)

    ROUTINGPTR:= JOBTPTR@.ROUTINGJ;
    WHILE ROUTINGPTR@.SERVER_GROUP_R <> SERV_GROUP DO
      RCUTINGPTR := ROUTINGPTR@.NEXTR;
    NEWEVENTPTR@.TIME_JOB_PART_TAKES := GENERATE_VAL
                      (ROUTINGPTR@.SERVICE_DIST,
                       ROUTINGPTR@.SERVICE_RATE);


    (* ATTATCH THE NEW EVENT RECORD TO THE JOB RECORD IF
       IT IS THE FIRST EVENT.  OTHERWISE ATTATCH IT TO THE
       LAST EVENT RECORD *)

    IF LASTEVENTPTR = NIL THEN
       NEWJOBPTR@.FIRST_JOB_PART := NEWEVENTPTR
    ELSE
       LASTEVENTPTR@.NEXT_JOB_PART := NEWEVENTPTR;

    (* DETERMINE THE SERVER GROUP NUMBER OF THE NEXT EVENT *)

    SERV_GROUP  := DESTINATION (ROUTINGPTR)

  UNTIL SERV_GROUP = 10;


(* ADD UP ALL THE SERVICE TIMES  FOR THE JOB EVENTS AND
   PUT THE VALUE INTO THE JOB PROCESSING_TIME FIELD  *)

  TOT_TIME:= 0;
  EVENTPTR := NEWJOBPTR@.FIRST_JOB_PART;
  WHILE EVENTPTR <> NIL DO
    BEGIN
      TOT_TIME := TOT_TIME + EVENTPTR@.TIME_JOB_PART_TAKES;
      EVENTPTR := EVENTPTR@.NEXT_JOB_PART
    END;
  NEWJOBPTR@.PROCESSING_TIME := TOT_TIME;
```

```
END;
(*-------------------------------------------------------------*)


(*-------------------------------------------------------------*)
(* INSERT_IN-QUEUE                                             *)
(*                                                             *)
(* CALLED BY: CREATE_JOB_STREAM                                *)
(*                                                             *)
(* INSERTS THE JOB RECORD  POINTED TO BY CURR                  *)
(* INTO THE HOLD QUEUE IN ASCENDING ORDER BY THE               *)
(* ARRIVAL_TIME FIELD OF THE JOB RECORD.  FIRST                *)
(* POINTS TO THE FIRST JOB RECORD IN THE HOLD QUEUE            *)
(*-------------------------------------------------------------*)

PROCEDURE INSERT_IN_QUEUE ( VAR CURR,FIRST : PTR);

VAR
  TEMP,PRED : PTR;

BEGIN

  IF FIRST = NIL THEN
    (* IF EMPTY MAKE IT THE FIRST RECORD*)
    BEGIN
      FIRST := CURR;
      FIRST@.NEXT_JOB := NIL
    END
  ELSE IF CURR@.ARRIVAL_TIME < FIRST@.ARRIVAL_TIME THEN
    (* IF CURR HAS THE LOWEST TIME INSERT AT THE FRONT*)
    BEGIN
      CURR@.NEXT_JOB := FIRST;
      FIRST := CURR
    END
  ELSE BEGIN

      (* FIND THE CORRECT LOCATION IN THE LIST AND INSERT*)
      TEMP := FIRST;
      WHILE (TEMP<>NIL)AND(CURR@.ARRIVAL_TIME >= TEMP@.ARRIVAL_TIME)DO
          BEGIN
            PREC := TEMP;
            TEMP := TEMP@.NEXT_JOB
          END;
      PREC@.NEXT_JOB := CURR;
      CURR@.NEXT_JOB := TEMP
    END
```

```
END;
(*-------------------------------------------------------------*)

BEGIN (* MAIN PROCEDURE FOR CREATE_JOB_STREAM *)
  (* UPLOAD THE DATA BASE RECORDS FOR THE MODEL NUM          *)
  BUILD_LL_FROM_DB (JOBTPTR);
  (* INITIAL LOOP CREATES ONE JOB FOR EACH JOB TYPE AND
     PLACES THIS JOB IN THE HOLD QUEUE BY ARRIVAL TIME *)
  TEMP_JTPTR:= JOBTPTR;
  HOLD := NIL;
  REPEAT
    NEWJOBPTR:= NIL;
    CREATE_JOB (TEMP_JTPTR, NEWJOBPTR);
    INSERT_IN_QUEUE (NEWJOBPTR, HOLD);
    TEMP_JTPTR:= TEMP_JTPTR@.NEXTJ
  UNTIL TEMP_JTPTR = NIL;
  (* NEXT LOOP TAKES THE FIRST JOB IN THE HOLD QUEUE AND
     ATTACHES IT TO THE END OF THE JOB ARRIVAL QUEUE.  IT
     CREATES A JOB OF THE JOB TYPE THAT WAS PUT IN THE
     ARRIVAL QUEUE AND INSERTS THAT JOB INTO THE HOLD
     QUEUE IN ORDER OF ARRIVAL TIME.  CYCLE IS CONTINUED
     UNTIL THE DESIRED NUMBER OF JOBS ARE CREATED *)

  FIRST_JOB := TRUE;
  JOBCOUNT := 0;

  REPEAT
    IF FIRST_JOB THEN
      BEGIN
        ARRIVEQPTR := HOLD;
        ENDQPTR := HOLD;
        FIRST_JOB := FALSE
      END
    ELSE
      BEGIN
        ENDQPTR@.NEXT_JOB := HOLD;
        ENDQPTR := HOLD
      END;

    HOLD := HOLD@.NEXT_JOB;
```

```
      ENDQPTR@.NEXT_JOB := NIL;
      TEMP_JTPTR:= JOBTPTR;
      WHILE TEMP_JTPTR@.JOB_TYPE_JT <> ENDQPTR@.JOB_TYPE DO
        TEMP_JTPTR := TEMP_JTPTR@.NEXTJ;
      CREATE_JOB  (TEMP_JTPTR, NEWJOBPTR);
      NEWJOBPTR@.ARRIVAL_TIME:= NEWJOBPTR@.ARRIVAL_TIME
          + ENDQPTR@.ARRIVAL_TIME;

      INSERT_IN_QUEUE (NEWJOBPTR, HOLD);
      JOBCOUNT:= JOBCOUNT + 1;
      ENDQPTR@.JOB_NUMB := JOBCOUNT

    UNTIL JOBCOUNT = TOTALJOBS

  END; (* MAIN PROCEDURE FOR CREATE_JOB_STREAM *)

  {*-------------------------------------------------------*}
  {*       END CREATE JOB STREAM MODULE                    *}
  {*-------------------------------------------------------*}


  {*-------------------------------------------------------*}
  {*-------------------------------------------------------*}
  {*         EXECUTE AND TABULATE MODULE                   *}
  {*-------------------------------------------------------*}
  {*-------------------------------------------------------*}



  {*-------------------------------------------------------*}
  {* PROCEDURE EXECUTE_AND_TABULATE                        *}
  {*                                                       *}
  {* CALLED BY:  CPMT MAIN DRIVER                          *}
  {*                                                       *}
  {* CALLS:  DEPART_FROM_SG, ARRIVE_AT_SG, INSERT_IN_      *}
  {* QUEUE, CREATE_SERVER-GROUPS,  STATS_FOR_JOBS,         *}
  {* STATS_FOR_JOB_TYPES, STATS_FOR_SERVER_GROUPS, AND     *}
  {* HIGHEST_JOB_TYPE (FUNCTION IN UPDATE MODULE)          *}
  {*                                                       *}
  {* THERE ARE THREE MAIN PARTS TO THE EXECUTE AND         *}
  {* TABULATE MODULE.  FIRST, THE PROCEDURE CALLS          *}
  {* THE CREATE_SERVER_GROUP PROCEDURE TO CREATE THE       *}
  {* LINKED LIST OF SERVER GROUP AND SERVER RECORDS        *}
  {* THROUGH WHICH THE JOBS WILL BE PROCESSED.  SECOND,    *}
  {* THE PROCEDURE EXECUTES THE MAIN PROCESSING LOOP TO    *}
  {* PROCESS ALL THE JOBS THROUGH THE SERVER GROUPS.       *}
  {* JOB PROCESSING IS HANDLED AS A SERIES OF JOB          *}
```

```
(* DEPARTURES AND JOB ARRIVALS WITH CALLS TO PROCEDURES *)
(* DEPART_FROM_SG AND ARRIVE_AT_SG.  THIRD, THE EXT     *)
(* CALCULATES THE STATISTICS FOR THE THE JOB TYPES AND  *)
(* SERVER GROUPS.                                       *)
(*------------------------------------------------------*)

PROCEDURE EXECUTE_AND_TABULATE (ARRIVEQPTR:PTR);

VAR

JPTR: PTR;
SGPTR: PTR;
SPTR: PTR;
FIRST_SGPTR :PTR;
MASTERQPTR :PTR;
EXITQPTR: PTR;
ENDEXITQ:PTR;
HIGHEST_JOB_TYPE :INTEGER;

CLK: INTEGER;
START_STATS_TIME :INTEGER;
END_STATS_TIME : INTEGER;
START_JOB_STATS :INTEGER;
END_JOBS_STATS: INTEGER;

(*------------------------------------------------------*)


(*------------------------------------------------------*)
(*   PROCEDURE STD_DEV                                  *)
(*                                                      *)
(*   CALCULATES THE STANDARD DEVIATION FOR              *)
(*   AVERAGE TIME IN QUEUE AND AVERAGE RESPONSE TIMES   *)
(*   FOR ALL JOBS                                       *)
(*                                                      *)
(*------------------------------------------------------*)


PROCEDURE STD_DEV (AVG_TIME_IN_Q, AVG_TIME_IN_SYS: REAL;
                   VAR STD_DEV_Q_TIME, STD_DEV_TIME_IN_SYS: REAL);

VAR

 TEMP : PTR;
 COUNT: INTEGER;
 Q_SUM_SQUARE, TIME_SUM_SQUARE: REAL;
```

```
BEGIN
 COUNT := 0;
 Q_SUM_SQUARE := 0.0;
 TIME_SUM_SQUARE := 0.0;
 TEMP := EXITCPTR;

 WHILE TEMP <> NIL DO

  BEGIN
    COUNT := COUNT + 1;
    Q_SUM_SQUARE := Q_SUM_SQUARE + SQR (AVG_TIME_IN_Q -
                                        TEMP@.TIME_IN_QUEUE);
    TIME_SUM_SQUARE := TIME_SUM_SQUARE + SQR (AVG_TIME_IN_SYS -
                                              TEMP@.TIME_IN_SYS);
    TEMP := TEMP@.NEXT_JOB;

  END; (* WHILE TEMP <> NIL *)

 STD_DEV_Q_TIME := SQRT (Q_SUM_SQUARE / (COUNT - 1) );
 STD_DEV_TIME_IN_SYS := SQRT (TIME_SUM_SQUARE / (COUNT - 1) )

END; (*PROCEDURE STD_DEV *)

(*-------------------------------------------------------------*)


(*-------------------------------------------------------------*)
(* PROCEDURE STD_DEV_JOB_TYPES                                 *)
(*                                                             *)
(* CALCULATES STANDARD DEVIATION FOR JOB RESPONSE TIMES        *)
(* AND QUEUE TIMES FOR JOBS OF A GIVEN JOB TYPE.               *)
(*                                                             *)
(*-------------------------------------------------------------*)


PROCEDURE STD_DEV_JOB_TYPES (I: INTEGER; AVG_TIME_IN_Q,
              AVG_TIME_IN_SYS: REAL; VAR STD_DEV_Q_TIME,
              STD_DEV_TIME_IN_SYS: REAL);

VAR

 TEMP: PTR;
 COUNT: INTEGER;
 Q_SUM_SQUARE, TIME_SUM_SQUARE: REAL;
```

```
BEGIN

  Q_SUM_SQUARE := 0;
  TIME_SUM_SQUARE := 0;
  COUNT:= 0;
  TEMP := EXITCPTR;
  WHILE TEMP <> NIL DO

    BEGIN
      IF TEMP@.JOB_TYPE = 1 THEN
        BEGIN (* COLLECTING STATISTICS FOR JOB TYPE 1 *)
          COUNT := COUNT + 1;
          Q_SUM_SQUARE := Q_SUM_SQUARE +
                          SQR (AVG_TIME_IN_Q - TEMP@.TIME_IN_QUEUE);
          TIME_SUM_SQUARE := TIME_SUM_SQUARE +
                          SQR (AVG_TIME_IN_SYS - TEMP@.TIME_IN_SYS);

        END; (* IF TEMP@.JOB_TYPE = 1 *)

      TEMP := TEMP@.NEXT_JOB;

    END; (* WHILE TEMP <> NIL *)

 IF COUNT <= 1 THEN
   BEGIN
     STD_DEV_TIME_IN_SYS := 0.0;
     STD_DEV_Q_TIME := 0.0;
   END (*IF COUNT <= 1 *)
 ELSE (* COUNT > 1 *)
   BEGIN
     STD_DEV_Q_TIME := SQRT (Q_SUM_SQUARE / (COUNT - 1) );
     STD_DEV_TIME_IN_SYS := SQRT (TIME_SUM_SQUARE / (COUNT - 1) )
   END (* ELSE COUNT > 1 *)

END; (* PROCEDURE STD_DEV_JOB_TYPES *)


(*-------------------------------------------------------*)
(* PROCEDURE STATS_FOR_JOBS                              *)
(*                                                       *)
(* THIS PROCEDURE TRAVERSES ALL JOB RECORDS AND          *)
(* COMPUTES THE MAXIMUN, MINIMUM AND AVERAGE STATS       *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE STATS_FOR_JOBS ;
```

```
VAR
 TEMP : PTR;
 COUNT : INTEGER;
 AVG_TIME_IN_SYS, AVG_TIME_IN_Q : REAL;
 MAX_TIME_IN_SYS, MAX_TIME_IN_Q : INTEGER;
 MIN_TIME_IN_SYS, MIN_TIME_IN_Q : INTEGER;
 TEMP_Q_TIME, TEMP_SYS_TIME : INTEGER;
 STD_DEV_Q_TIME, STD_DEV_TIME_IN_SYS: REAL;

BEGIN

  COUNT := 0;
  TEMP := EXITPTR;
  TEMP_SYS_TIME := 0;
  TEMP_Q_TIME := 0;
  MAX_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
  MIN_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
  MAX_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
  MIN_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;

  WHILE TEMP <> NIL DO

    BEGIN

      TEMP_SYS_TIME := TEMP_SYS_TIME + TEMP@.TIME_IN_SYS;
      TEMP_Q_TIME := TEMP_Q_TIME + TEMP@.TIME_IN_QUEUE;
      IF TEMP@.TIME_IN_QUEUE > MAX_TIME_IN_Q THEN
        MAX_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
      IF TEMP@.TIME_IN_QUEUE < MIN_TIME_IN_Q THEN
        MIN_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
      IF TEMP@.TIME_IN_SYS > MAX_TIME_IN_SYS THEN
        MAX_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
      IF TEMP@.TIME_IN_SYS < MIN_TIME_IN_SYS THEN
        MIN_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
      COUNT := COUNT +1;
      TEMP := TEMP@.NEXT_JOB;

    END; (* WHILE TEMP <> NIL *)

  AVG_TIME_IN_SYS := TEMP_SYS_TIME / COUNT;
  AVG_TIME_IN_Q := TEMP_Q_TIME / COUNT;
  STD_DEV (AVG_TIME_IN_Q, AVG_TIME_IN_SYS,
                          STD_DEV_Q_TIME, STD_DEV_TIME_IN_SYS);

  WRITELN (OUTFILE, 'NUMB JOBS IS: ',COUNT);
```

```
  WRITELN (OUTFILE); WRITELN (OUTFILE);
  WRITELN (OUTFILE, 'JOB      MAX     MIN     MEAN    STDD ',
           '      MAX     MIN     MEAN      STDD');

  WRITELN (OUTFILE, 'TYPE     QTIME   QTIME   QTIME   QTIME',
           '     STIME   STIME   STIME     STIME');
  WRITELN (OUTFILE);
  WRITELN (OUTFILE,
           'ALL',
           MAX_TIME_IN_Q:8,
           MIN_TIME_IN_Q:8,
           AVG_TIME_IN_Q:8:3,
           STD_DEV_Q_TIME:8:3,
           MAX_TIME_IN_SYS:7,
           MIN_TIME_IN_SYS:6,
           AVG_TIME_IN_SYS:8:3,
           STD_DEV_TIME_IN_SYS:8:3);

END;  (* PROCEDURE STATS_FOR_JOBS *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE STATS_FOR_SERVER_GROUPS                     *)
(*                                                       *)
(* THIS PROCEDURE TRAVERSES ALL THE SERVER GROUP AND     *)
(* SERVER RECORDS AND COMPUTES THE SG STATISTICS         *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE STATS_FOR_SERVER_GROUPS;

VAR
 TEMP, TEMPS : PTR;
 COUNT : INTEGER;
 AVG_Q_LENGTH, UTILIZATION : REAL;
 TOTAL_SYSTEM_TIME : REAL;
 SGUTIL:REAL;
 SYS_STOP_TIME : INTEGER;

BEGIN

  COUNT := 0;
  TEMP := FIRST_SGPTR@.NEXT_SERVER_GROUP;
```

```
  TOTAL_SYSTEM_TIME := CLK;

(* NOTE TOTAL_SYSTEM_TIME IS NOW THE TIME THE SYSTEM OPERATED     *)
 WRITELN (OUTFILE); WRITELN (OUTFILE);
 WRITELN (OUTFILE, 'SERVER   MAX      MIN      AVG     ',
          '   SG      SERVER   SERVER'');
 WRITELN (OUTFILE, 'GROUP    QLEN     QLEN     QLEN    ',
          '   UTIL    NUMB     UTIL');

 WHILE TEMP <> NIL DO

   BEGIN

     IF TEMP@.Q_LEN_VECTOR <> 0 THEN
       AVG_Q_LENGTH:= TEMP@.Q_LEN_VECTOR / TOTAL_SYSTEM_TIME
     ELSE
       AVG_Q_LENGTH:= 0;
     UTILIZATION:= TEMP@.CUM_BUSY_TIME / TOTAL_SYSTEM_TIME;

     WRITELN (OUTFILE);
     WRITELN (OUTFILE,
              TEMP@.SERVER_GROUP:4,
              TEMP@.MAX_Q_LENGTH:7,
              TEMP@.MIN_Q_LENGTH:7,
              AVG_Q_LENGTH:10:3,
              UTILIZATION:10:3);

     TEMPS:= TEMP@.FIRST_SERVER;

     WHILE TEMPS <> NIL DO
       BEGIN

         SGUTIL:= TEMPS@.S_CUM_BUSY/TOTAL_SYSTEM_TIME;
         WRITELN (OUTFILE,'                                          ',
                  TEMPS@.SERVER:7,
                  SGUTIL:11:3);
         TEMPS:= TEMPS@.NEXT_SERVER;

       END;

    TEMP := TEMP@.NEXT_SERVER_GROUP;

  END; (* WHILE TEMP <> NIL *)

END;  (* PROCEDURE STATS_FOR_SERVER_GROUPS *)

(*-------------------------------------------------*)
```

```
(*-------------------------------------------------------------*)
(* PROCEDURE STATS_FOR-JOB_TYPES                               *)
(*                                                             *)
(* THIS PROCEDURE TRAVERSES ALL JOB RECORDS, AND               *)
(* COMPUTES THE MAXIMUM, MINIMUM, AND AVERAGE STATS            *)
(* BY JOB TYPE.                                                *)
(*                                                             *)
(*-------------------------------------------------------------*)

PROCEDURE STATS_FOR_JOB_TYPES ;

VAR

 TEMP : PTR;
 I, COUNT : INTEGER;
 AVG_TIME_IN_SYS, AVG_TIME_IN_Q : REAL;
 MAX_TIME_IN_SYS, MAX_TIME_IN_Q : INTEGER;
 MIN_TIME_IN_SYS, MIN_TIME_IN_Q : INTEGER;
 TEMP_Q_TIME, TEMP_SYS_TIME : INTEGER;
 STD_DEV_Q_TIME, STD_DEV_TIME_IN_SYS: REAL;
 INITIALIZED : BOOLEAN;

BEGIN  (* PROCEDURE STATS FOR JOB TYPES *)

  FOR I := 1 TO HIGHEST_JOB_TYPE DO

    BEGIN

      COUNT := 0;
      TEMP := EXITQPTR;
      INITIALIZED := FALSE;
      WHILE TEMP <> NIL DO

        BEGIN
          IF TEMP@.JOB_TYPE = I THEN
            (* COLLECT STATISTICS FOR JOB TYPE I *)

            BEGIN

              IF NOT INITIALIZED THEN
                BEGIN  (* IF NOT INITIALIZED *)
                  TEMP_SYS_TIME := 0;
                  TEMP_Q_TIME := 0;
                  MAX_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
                  MIN_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
```

```
                    MAX_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
                    MIN_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
                    INITIALIZED:= TRUE;
                  END; (* IF NOT INITIALIZED *)

                TEMP_SYS_TIME := TEMP_SYS_TIME + TEMP@.TIME_IN_SYS;
                TEMP_Q_TIME := TEMP_Q_TIME + TEMP@.TIME_IN_QUEUE;
                IF TEMP@.TIME_IN_QUEUE > MAX_TIME_IN_Q THEN
                   MAX_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
                IF TEMP@.TIME_IN_QUEUE < MIN_TIME_IN_Q THEN
                   MIN_TIME_IN_Q := TEMP@.TIME_IN_QUEUE;
                IF TEMP@.TIME_IN_SYS > MAX_TIME_IN_SYS THEN
                   MAX_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
                IF TEMP@.TIME_IN_SYS < MIN_TIME_IN_SYS THEN
                   MIN_TIME_IN_SYS := TEMP@.TIME_IN_SYS;
                COUNT := COUNT +1

              END; (* IF TEMP@.JOB_TYPE = I *)

            TEMP := TEMP@.NEXT_JOB;

          END; (* WHILE TEMP <> NIL *)

        IF COUNT = 0 THEN
         AVG_TIME_IN_SYS := 0
        ELSE
         AVG_TIME_IN_SYS := TEMP_SYS_TIME / (COUNT);

        IF COUNT = 0 THEN

         BEGIN

           STD_DEV_Q_TIME := 0.0;
           STD_DEV_TIME_IN_SYS := 0.0;
           AVG_TIME_IN_Q := 0

         END (* IF COUNT = 0 *)

        ELSE

         BEGIN

           AVG_TIME_IN_Q := TEMP_Q_TIME / (COUNT);
           STD_DEV_JOB_TYPES (I, AVG_TIME_IN_Q, AVG_TIME_IN_SYS,
                              STD_DEV_Q_TIME, STD_DEV_TIME_IN_SYS)
         END; (* ELSE COUNT > 0 *)

        WRITELN (OUTFILE);
```

```
  WRITELN (OUTFILE,
           I:3,
           MAX_TIME_IN_Q:8,
           MIN_TIME_IN_Q:8,
           AVG_TIME_IN_Q:8:3,
           STD_DEV_Q_TIME:8:3,
           MAX_TIME_IN_SYS:7,
           MIN_TIME_IN_SYS:6,
           AVG_TIME_IN_SYS:8:3,
           STD_DEV_TIME_IN_SYS:8:3);

 END (*FOR I := 1 TO HIGHEST_JOB_TYPE DO *)

END;  (* PROCEDURE STATS_FOR_JOB_TYPES *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE  CREATE_SERVER_GROUPS                       *)
(*                                                       *)
(* CALLED BY: EXECUTE_AND_TABULATE                       *)
(*                                                       *)
(* THIS PROCEDURE CREATES THE LINKED LIST OF SERVER      *)
(* GROUP AND SERVER RECORDS.  THE NUMBER OF SERVERS      *)
(* IN EACH SERVER GROUP CREATED IS ACCESSED FROM THE     *)
(* NUM_SERVERS ARRAY WHICH WAS UPLOADED FROM THE DB      *)
(* SERVER GROUP RECORD BY THE BUILD_LL_FROM_DB           *)
(* PROCEDURE.  THE LINKED LIST OF SERVER GROUP RECORDS   *)
(* IS POINTED TO BY THE FIRST_SGPTR POINTER.             *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE CREATE_SERVER_GROUPS;

VAR

SGTEMP :PTR;
STEMP :PTR;
JTEMP : PTR;
LAST_SGTEMP : PTR;
LAST_STEMP: PTR;
I,J :INTEGER;


BEGIN (* CREATE_SERVER_GROUPS *)
```

```
(*CREATE THE ARRIVALQUE SERVER RECORD *)
(* FIRST JOB IN ARRIVAL Q BECOMES FIRST EVENT
(* IN THE MASTER EVENT LIST AND THE ARRIVAL Q
(* IS ATTATCHED TO THE 0 SERVER GROUP RECORD *)

NEW (SGTEMP, SERVER_GROUP_RECORD);
SGTEMP@.TAG := SERVER_GROUP_RECORD;
SGTEMP@.SERVER_GROUP:= 0;
FIRST_SGPTR := SGTEMP;
FIRST_SGPTR@.NEXT_SERVER_GROUP:= NIL;
MASTERQPTR := SGTEMP;
MASTERQPTR@.NEXT_MASTQ := NIL;
JTEMP:=ARRIVEQPTR;
FIRST_SGPTR@.FIRST_IN_Q := JTEMP;
MASTERQPTR@.NEXT_EVENT_TIME:= JTEMP@.ARRIVAL_TIME;
LAST_SGTEMP := SGTEMP;

(* NOW CREATE THE SERVER GROUP AND SERVER
   RECORDS FOR THE NORMAL SERVERS AND LINK THEM
   INTO THE LIST OF SERVER GROUP RECORDS *)


FOR I := 1 TO MAX_SERV DO
  IF NUM_SERVERS  I  <> 0 THEN
    BEGIN

      (* CREATE AND INITIALIZE THE SERVER REC *)
      NEW (SGTEMP, SERVER_GROUP_RECORD);
      SGTEMP@.TAG := SERVER_GROUP_RECORD;
      SGTEMP@.SERVER_GROUP := I;
      SGTEMP@.FIRST_SERVER := NIL;
      SGTEMP@.NEXT_SERVER_GROUP := NIL;
      SGTEMP@.FIRST_IN_Q := NIL;
      SGTEMP@.NEXT_MASTQ := NIL;
      SGTEMP@.NEXT_S_EVENT := NIL;
      SGTEMP@.Q_LEN_VECTOR := 0;
      SGTEMP@.Q_LEN_TIME := 0;
      SGTEMP@.MAX_Q_LENGTH := 0;
      SGTEMP@.MIN_Q_LENGTH := 0;
      SGTEMP@.START_BUSY_TIME := 0;
      SGTEMP@.STOP_BUSY_TIME := 0;
      SGTEMP@.CUM_BUSY_TIME := 0;
      SGTEMP@.NEXT_EVENT_TIME := 0;

      IF I = 1 THEN
         BEGIN
            FIRST_SGPTR@.NEXT_SERVER_GROUP := SGTEMP;
            LAST_SGTEMP:= SGTEMP;
```

```
            END
          ELSE
            BEGIN

              LAST_SGTEMP@.NEXT_SERVER_GROUP:= SGTEMP;

              LAST_SGTEMP := SGTEMP;
            END;
          (* CREATE, INITIALIZE, ATTATCH THE SERVER RECS *)

          FOR J := 1 TO NUM_SERVERS  I  DO
            BEGIN
              NEW (STEMP, SERVER_RECORD);
              STEMP@.TAG := SERVER_RECORD;
              STEMP@.SERVER:= J;
              STEMP@.JOB_NO := NIL;
              STEMP@.TIME_EXIT := 0;
              STEMP@.NEXT_SERVER := NIL;
              STEMP@.BUSY := FALSE;
              IF J = 1 THEN
                SGTEMP@.FIRST_SERVER := STEMP
              ELSE
                LAST_STEMP@.NEXT_SERVER := STEMP;
              LAST_STEMP :=STEMP;
            END; (* FOR J = 1 TO NUM_SERVERS *)

        END; (* IF NUM SERVERS <> ) *)

END; (* PROCEDURE CREATE SERVER GROUPS *)

(*-------------------------------------------------------------*)
(* PROCEDURE DEPART_FROM_SG                                    *)
(*                                                             *)
(* CALLED BY: EXECUTE_AND_TABULATE                             *)
(*                                                             *)
(* CALLS:  JOE_ARRIVAL, NO_JOB_IN_SG_Q, JOB_IN_SG_Q            *)
(*                                                             *)
(* THIS PROCEDURE PROCESSES THE NEXT EVENT IN THE              *)
(* MASTER EVENT Q, WHICH IS A JOB DEPARTURE.                   *)
(* IT DETACHES THE FIRST SERVER_GROUP FROM THE MASTER          *)
(* EVENT_QUEUE AND DETACHES THE JOB WHICH HAS JUST             *)
(* FINISHED PROCESSING FROM THE SERVER GROUP.  THE             *)
(* DETACHED SERVER GROUP IS POINTED TO BY SGPTR AND            *)
(* THE DETACHED JOB IS POINTED TO BY JPTR.                     *)
(* AFTER THE JOB IS DETACHED THE SERVER GROUP IS               *)
(* UPDATED DEPENDING UPON CIRCUMSTANCES.  THERE ARE            *)
(* THREE CONDITIONS OF SERVER GROUP UPDATE. THE FIRST          *)
```

```
(* CASE IS THE CASE OF THE ARRIVAL SG.  IF THE JOB       *)
(* DEPARTED FROM THE ARRIVAL SERVER GROUP, THEN          *)
(* THE ARRIVAL QUEUE IS BUMPED UP AND SG STATS UPDATED   *)
(* FOR THE REVISED NEXT EVENT TIME.                      *)
(* THE SECOND CASE IS IF THE SG OF THE DEPARTING JOB     *)
(* IS NOT SG 0 AND HAS NO SERVER GROUP Q.  THE THIRD     *)
(* CASE IS IF THE SG OF THE DEPARTING JOB IS NOT SG 0    *)
(* AND HAS A SERVER GROUP Q.  THE PROCESSING WHICH       *)
(* TAKES PLACE FOR EACH CASE IS DISCUSSED UNDER          *)
(* APPROPRIATE PROCESSING PROCEDURES.                    *)
(*-------------------------------------------------------*)

PROCEDURE DEPART_FROM_SG;

VAR

TEMP_SPTR : PTR;
NJPTR: PTR;
ATTATCH_TIME: INTEGER;
DEPARTCASE: INTEGER;

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE JOB_ARRIVAL                                 *)
(*                                                       *)
(* CALLED BY:  DEPART_FROM_SG                            *)
(*                                                       *)
(* THE JOB IS DEPARTING FROM SG 0 SO IT IS AN ARRIVAL    *)
(* INTO THE SYSTEM.  THE JOB ARRIVING AFTER THE CURRENT  *)
(* DEPARTING JOB IS MOVED UP IN THE QUEUE.               *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE JOB_ARRIVAL;

BEGIN (* PROCEDURE JOB_ARRIVAL *)

  JPTR:= SGPTR@.FIRST_IN_Q;
  SGPTR@.FIRST_IN_Q := JPTR@.NEXT_JOB;
  JPTR@.NEXT_JOB := NIL;
  JPTR@.SERVING_EVENT:= JPTR@.FIRST_JOB_PART;
  IF SGPTR@.FIRST_IN_Q = NIL THEN
    ATTATCH_TIME:= 0
  ELSE
    ATTATCH_TIME:= SGPTR@.FIRST_IN_Q@.ARRIVAL_TIME;
  SGPTR@.NEXT_EVENT_TIME:= ATTATCH_TIME;
```

```
END; (* PROCEDURE JOB_ARRIVAL *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE NO_JOB_IN_SG_Q                              *)
(*                                                       *)
(* THERE IS NO JOB IN THE SG Q WAITING TO BE PROCESSED.  *)
(* THE PROCEDURE UPDATES THE SERVER RECORD STATISTICS    *)
(* FOR THE SERVER RECORD WHICH THE DEPARTING JOB JUST    *)
(* VACATED.                                              *)
(*                                                       *)
(*-------------------------------------------------------*)


PROCEDURE NO_JOB_IN_SG_Q;

BEGIN (* IF NJPTR *)

  (* THERE IS NO JOB IN THE SG Q *)
  (* THEREFORE UPDATE THE STATS ON SERVER RECS*)

  SPTR@.S_CUM_BUSY := SPTR@.S_CUM_BUSY +
    (CLK - SPTR@.S_START_BUSY);
  SPTR@.S_START_BUSY := 0;
  SPTR@.BUSY := FALSE;
  SPTR@.TIME_EXIT := 0;
  SPTR@.JOB_NO := NIL;

END; (* PROCEDURE NO JOB IN SG Q *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE JOB_IN_SG_Q                                 *)
(*                                                       *)
(* CALLED BY: DEPART_FROM_SG                             *)
(*                                                       *)
(* THERE IS A JOB IN THE QUEUE WAITING FOR AN AVAILABLE  *)
(* SERVER.  THE JOB IS ATTACHED TO THE SERVER JUST       *)
(* VACATED BY THE DEPARTING JOB.  THE QUEUE STATISTICS   *)
(* IN THE SERVER GROUP RECORD ARE UPDATED TO REFLECT     *)
(* THE FACT THAT THERE IS ONE LESS JOB IN THE QUEUE.     *)
(*                                                       *)
```

```
(*-------------------------------------------------------*)

PROCEDURE JOB_IN_SG_Q;

BEGIN (* PROCEDURE JOB IN SG Q *)

  SGPTR@.FIRST_IN_Q:= NJPTR@.NEXT_JOB;
  NJPTR@.NEXT_JOB := NIL;

  (* UPDATE Q STATISTICS *)

  SGPTR@.Q_LEN_VECTOR := SGPTR@.Q_LEN_VECTOR +
     ((CLK - SGPTR@.Q_LEN_TIME) * SGPTR@.Q_LENGTH);
  SGPTR@.Q_LENGTH := SGPTR@.Q_LENGTH - 1;
  SGPTR@.Q_LEN_TIME := CLK;

  (* ATTATCH NEXT JOB TO THE SERVER *)
  (* UPDATE THE SERVER STATISTICS *)

  SPTR@.JOB_NO := NJPTR;
  SPTR@.TIME_EXIT := CLK +
    NJPTR@.SERVING_EVENT@.TIME_JOB_PART_TAKES;

END; (* PROCEDURE JOB IN SG Q *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE FIND_NEXT_EVENT_TIME                        *)
(*                                                       *)
(* CALLED BY: DEPART_FROM_SG                             *)
(*                                                       *)
(* IF THE JOB IS CASE 2 OR 3 DEPARTURE (NOT DEPARTING    *)
(* FROM SG 0) THEN AFTER EITHER THE NO_JOB_IN_SG_Q OR    *)
(* THE JOB_IN_SG_Q PROCEDURES ARE EXECUTED, THIS         *)
(* PROCEDURE IS CALLED TO TRAVERSE ALL THE SERVER GROUP  *)
(* SERVER RECORDS TO FIND THE NEXT EVENT TIME FOR THE    *)
(* SERVER GROUP AND TO UPDATE THE SERVER GROUP RECORD    *)
(*                                                       *)
(*-------------------------------------------------------*)

PROCEDURE FIND_NEXT_EVENT_TIME;

BEGIN

      (* NOW FIND THE NEXT EVENT TIME FOR THE SERVER GROUP *)
```

```
        SPTR:= SGPTR@.FIRST_SERVER;
        TEMP_SPTR := SPTR;
        ATTATCH_TIME := SPTR@.TIME_EXIT;
        WHILE SPTR@.NEXT_SERVER <> NIL DO
          BEGIN
            SPTR:= SPTR@.NEXT_SERVER;
            IF ((ATTATCH_TIME = 0) OR
                 (SPTR@.TIME_EXIT < ATTATCH_TIME)) THEN
              BEGIN (* IF ATTATCH *)
                ATTATCH_TIME := SPTR@.TIME_EXIT;
                TEMP_SPTR := SPTR;
              END; (* IF ATTATCH *)
          END; (* WHILE SPTR *)

        SGPTR@.NEXT_EVENT_TIME := ATTATCH_TIME;
        IF ATTATCH_TIME = 0 THEN
          BEGIN
            (* SERVER GROUP GOES FROM BUSY TO IDLE *)
            (* UPDATE SERVER STATS TO REFLECT CHANGE *)

            SGPTR@.NEXT_S_EVENT := NIL;
            SGPTR@.CUM_BUSY_TIME := SGPTR@.CUM_BUSY_TIME +
              (CLK - SGPTR@.START_BUSY_TIME);
            SGPTR@.START_BUSY_TIME := 0;

          END  (* IF ATTATCH *)
        ELSE
          SGPTR@.NEXT_S_EVENT:= TEMP_SPTR;

      END; (* PROCEDURE FIND NEXT EVENT TIME *)

    (*-------------------------------------------------------------*)
    (*-------------------------------------------------------------*)

    BEGIN  (* PROCEDURE DEPART_FROM_SG *)

      (* DETATCH NEXT SERVER GROUP FROM MASTERQ *)

      SGPTR:= MASTERQPTR;
      MASTERQPTR := MASTERQPTR@.NEXT_MASTQ;
      SGPTR@.NEXT_MASTQ:= NIL;

      (* UPDATE CLOCK TO CURRENT TIME *)
      CLK := SGPTR@.NEXT_EVENT_TIME;
```

```
    (* DETATCH THE DEPARTING JOB FROM THE
       SERVER GROUP AND UPDATE THE SERVER
       GROUP *)

    IF SGPTR@.SERVER_GROUP = 0 THEN
      DEPARTCASE:= 1

    ELSE
      BEGIN (* ELSE *)

        (* DETATCH THE DEPARTING JOB *)
        SPTR:= SGPTR@.NEXT_S_EVENT;
        JPTR := SPTR@.JOB_NO;
        JPTR@.SERVING_EVENT := JPTR@.SERVING_EVENT@.NEXT_JOB_PART;


        NJPTR := SGPTR@.FIRST_IN_Q;
        IF NJPTR = NIL THEN
          DEPARTCASE:= 2

        ELSE
          DEPARTCASE:= 3
      END;

    CASE DEPARTCASE OF

      1: JOB_ARRIVAL;

      2: NO_JOB_IN_SG_Q;

      3: JOB_IN_SG_Q;

    END; (* CASE DEPARTCASE *)

    IF (DEPARTCASE = 2) OR (DEPARTCASE = 3 ) THEN
      FIND_NEXT_EVENT_TIME;

  END; (* PROCEDURE DEPARTURE *)

  (*-------------------------------------------------*)


  (*-------------------------------------------------*)
  (* INSERT_EVENT : INSERTS THE EVENT_RECORD POINTED TO BY  *)
  (*    CURR INTO THE EVENT LIST. FIRST POINTS TO THE       *)
```

```
(*     FIRST RECORD IN THE EVENT LIST.                              *)
(*-------------------------------------------------------------*)
PROCEDURE INSERT_IN_QUEUE (   SGPTR: PTR;
                              VAR  MASTERQPTR : PTR);

VAR
  TEMP,PRED : PTR;

BEGIN

    IF MASTERQPTR = NIL THEN
    (* IF EMPTY MAKE IT THE FIRST RECORD*)
    BEGIN
      MASTERQPTR := SGPTR;
      MASTERQPTR @.NEXT_MASTQ := NIL
    END
  ELSE IF SGPTR@.NEXT_EVENT_TIME< MASTERQPTR@.NEXT_EVENT_TIME THEN
    (* IF SGPTR HAS THE LOWEST TIME  INSERT AT THE FRONT*)
    BEGIN
      SGPTR@.NEXT_MASTQ := MASTERQPTR ;
      MASTERQPTR := SGPTR;
    END
  ELSE BEGIN

      (* FIND THE CORRECT LOCATION IN THE LIST AND INSERT*)
      TEMP := MASTERQPTR ;
      WHILE (TEMP<>NIL)AND(SGPTR@.NEXT_EVENT_TIME >=
          TEMP@.NEXT_EVENT_TIME)DO
          BEGIN
            PRED := TEMP;
            TEMP := TEMP@.NEXT_MASTQ;
          END;
      PRED@.NEXT_MASTQ := SGPTR;
      SGPTR@.NEXT_MASTQ := TEMP
    END
END;


(*-------------------------------------------------------------*)
(* PROCEDURE SG_Q_INSERT.                                      *)
(*                                                             *)
(* THIS PROCEDURE INSERTS THE JOB POINTED TO BY THE            *)
(* JPTR POINTER INTO THE SERVER GROUP QUEUE OF THE             *)
(* SERVER GROUP POINTED TO BY SGPTR.                           *)
(*                                                             *)
(*-------------------------------------------------------------*)

PROCEDURE SG_Q_INSERT (VAR SGPTR,JPTR: PTR);
```

```
VAR TEMP:PTR;

BEGIN
  TEMP:= SGPTR@.FIRST_IN_Q;

  WHILE TEMP@.NEXT_JOB <> NIL DO
    TEMP:= TEMP@.NEXT_JOB;
  TEMP@.NEXT_JOB:=JPTR;

END; (* PROCEDURE SG_Q_INSERT *)
(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(* PROCEDURE ARRIVE_AT_SG                                *)
(*                                                       *)
(* CALLED BY: EXECUTE_AND_TABULATE                       *)
(*                                                       *)
(* CALLS: JOB_COMPLETED, ATTACH_FIRST_IN_Q               *)
(*        ATTACH_JOB_TO_SERVER, INSERT_IN_SG_Q           *)
(*                                                       *)
(* THIS PROCEDURE HANDLES THE ARRIVAL OF THE JOB         *)
(* POINTED TO BY JPTR TO A SERVER GROUP AND UPDATES      *)
(* THE SERVER GROUP/JOB STATISTICS AS NECESSARY AFTER    *)
(* THE ARRIVAL.  THERE ARE 4 ARRIVAL CASES.              *)
(* IN THE FIRST CASE THE JOB HAS FINISHED PROCESSING     *)
(* AND IS ARRIVING AT THE EXIT SERVER GROUP.  THE  2ND   *)
(* CASE IS FOR A JOB WHICH IS NOT COMPLETE AND IS        *)
(* ARRIVING AT A SERVER GROUP THAT HAS A QUEUE.          *)
(* IT GETS ADDED TO THE QUEUE ACCORDING TO THE QUEUE     *)
(* DISCIPLINE. THE THIRD CASE IS A JOB WHICH IS          *)
(* ARRIVING TO A SERVER GROUP WITH AN AVAILABLE SERVER.  *)
(* THE 4TH CASE IS A JOB ARRIVING TO A SG WITH NO        *)
(* AVAIL SERVER AND NO Q.  IT GETS ADDED TO SG AS FIRST  *)
(* IN Q.                                                 *)
(*-------------------------------------------------------*)

PROCEDURE ARRIVE_AT_SG;

VAR

AVAIL_SERVER: BOOLEAN;
TEMP, PRED: PTR;
```

```
INMASTQ:BOOLEAN;
ARRIVECASE: INTEGER;

(*-------------------------------------------------------*)
(* PROCEDURE JOB_COMPLETE                                *)
(*                                                       *)
(* CALLED BY:  ARRIVE_AT_SG                              *)
(*                                                       *)
(* THE JOB POINTED TO BY JPTR IS COMPLETE AND IS         *)
(* EXITING THE SYSTEM.  THE JOB STATS ARE UPDATED AND    *)
(* THE JOB IS ADDED TO THE EXIT QUEUE POINTED TO BY      *)
(* EXITQPTR.                                             *)
(*-------------------------------------------------------*)

PROCEDURE JOB_COMPLETED;

BEGIN (* JOB_COMPLETED *)

  JPTR@.TIME_IN_SYS:= CLK - JPTR@.ARRIVAL_TIME;
  JPTR@.TIME_IN_QUEJE := JPTR@.TIME_IN_SYS -
    JPTR@.PROCESSING_TIME;
  JPTR@.EXIT_SYS_TIME:=CLK;


    (* JOB IS COMPLETE, ADD IT TO EXIT Q *)

  IF EXITQPTR = NIL THEN
    EXITQPTR:=JPTR
  ELSE
    ENDEXITQ@.NEXT_JOB:= JPTR;
  ENDEXITQ:=JPTR;

END;  (* JOB  COMPLETED *)

(*-------------------------------------------------------*)
(* PROCEDURE ATTACH_JOB_TO_SERVER                        *)
(*                                                       *)
(* CALLED BY: ARRIVE_AT_SG                               *)
(*                                                       *)
(* THE JOB POINTED TO BY JPTR IS ATTACHED TO THE AVAIL   *)
(* SERVER POINTED TO BY SPTR.   THE SERVER STATISTICS    *)
(* ARE UPDATED.  THE SERVER GROUP STATISTICS ARE UP-     *)
(* DATED DEPENDING ON WHETHER SERVER GROUP WAS IDLE OR   *)
(* BUSY WHEN JOB ARRIVED.  IF THE ARRIVAL SERVER GROUP   *)
(* WAS IN THE MASTER EVENT Q IT IS DETACHED FROM         *)
```

```
(* THE MASTER EVENT Q.  IT IS REINSERTED INTO THE MASTER*)
(* EVENT Q BY ITS CURRENT EVENT_TIME , WHICH MAY HAVE   *)
(* CHANGED.                                             *)
(*------------------------------------------------------*)

PROCEDURE ATTACH_JOB_TO_SERVER;

  BEGIN (* ATTATCH JOB TO SERVER *)

     SPTR@.JOB_NO := JPTR;
     SPTR@.TIME_EXIT := CLK +
       JPTR@.SERVING_EVENT@.TIME_JOB_PART_TAKES;
     SPTR@.BUSY := TRUE;
     SPTR@.S_START_BUSY := CLK;

     IF SGPTR@.NEXT_EVENT_TIME = 0 THEN
       BEGIN (* SERVER GROUP WAS IDLE *)
         SGPTR@.NEXT_EVENT_TIME := SPTR@.TIME_EXIT;
         SGPTR@.NEXT_S_EVENT:= SPTR;
         SGPTR@.START_BUSY_TIME := CLK;
       END (* SERVER GROUP WAS IDLE *)
     ELSE
       BEGIN (* SERVER GROUP WAS BUSY *)
         IF SPTR@.TIME_EXIT < SGPTR@.NEXT_EVENT_TIME THEN
           BEGIN
             SGPTR@.NEXT_S_EVENT := SPTR;
             SGPTR@.NEXT_EVENT_TIME :=
             SPTR@.TIME_EXIT;

           END;

       END; (* SERVER GROUP WAS BUSY *)

     IF MASTERQPTR <> NIL THEN
       BEGIN (* IF MASTERQPTR *)

         IF SGPTR@.SERVER_GROUP = MASTERQPTR@.SERVER_GROUP THEN
           MASTERQPTR:= SGPTR@.NEXT_MASTQ
         ELSE
           BEGIN  (* ELSE NOT FIRST IN Q*)
             INMASTQ:= FALSE;
             TEMP:= MASTERQPTR;
             WHILE (TEMP <> NIL) AND
                       (INMASTQ = FALSE) DO
               BEGIN (* WHILE TEMP *)
                 IF TEMP@.SERVER_GROUP =
                     SGPTR@.SERVER_GROUP THEN
                     INMASTQ:= TRUE
```

```
                    ELSE
                      BEGIN (* ELSE*)
                        PRED:= TEMP;
                        TEMP:= TEMP@.NEXT_MASTQ;
                      END; (* ELSE *)
                  END; (* WHILE TEMP *)

              IF INMASTQ THEN
                PRED@.NEXT_MASTQ:= TEMP@.NEXT_MASTQ;
          END; (* ELSE  NOT FIRST IN Q *)
      END; (*   *)
      IF SGPTR@.NEXT_EVENT_TIME <> 0 THEN
      INSERT_IN_QUEUE (SGPTR, MASTERQPTR);

END; (* ATTATCH JOB TO SERVER *)

(*-------------------------------------------------------*)
(* PROCEDURE ATTACH_FIRST_IN_Q                           *)
(*                                                       *)
(* CALLED BY: ARRIVE_AT_SG                               *)
(*                                                       *)
(* PROCEDURE ATTACHES THE JOB POINTED TO BY JPTR TO THE  *)
(* SG FIRST_IN_Q POINTER AND UPDATES THE SERVER GROUP    *)
(* Q STATISTICS.  PROCEDURE IS CALLED WHEN JOB IS        *)
(* LINKED TO A SERVER THAT PREVIOUSLY HAD NO QUEUE BUT   *)
(* ALL THE SG SERVERS ARE CURRENTLY BUSY.                *)
(*-------------------------------------------------------*)

PROCEDURE ATTACH_FIRST_IN_Q;

BEGIN  (* ATTATCH JOB TO FIRST IN Q *)

  SGPTR@.FIRST_IN_Q := JPTR;
  SGPTR@.Q_LENGTH:= SGPTR@.Q_LENGTH + 1;
  SGPTR@.Q_LEN_TIME := CLK;
  IF SGPTR@.Q_LENGTH > SGPTR@.MAX_Q_LENGTH THEN
  SGPTR@.MAX_Q_LENGTH:= SGPTR@.Q_LENGTH;

END; (* ATTATCH JOB TO FIRST IN Q *)


(*-------------------------------------------------------*)
(* PROCEDURE INSERT_IN_SG_Q                              *)
(*                                                       *)
(* CALLS : SG_Q_INSERT                                   *)
(*                                                       *)
(* CALLED BY : ARRIVE_AT_SG                              *)
(*                                                       *)
```

```
(* IF THE JOB ARRIVES AT A SERVER GROUP THAT HAS A Q     *)
(* IT IS INSERTED INTO THE QUEUE IN AN ORDER DEPENDING   *)
(* ON THE QUEUEING DISCIPLINE.  THE SERVER GROUP         *)
(* QUEUE STATS ARE UPDATED TO REFLECT THE ADDED JOB.     *)
(*-------------------------------------------------------*)

PROCEDURE INSERT_IN_SG_Q;

BEGIN

  SG_Q_INSERT  (SGPTR,JPTR);
  (*UPDATE Q STATS *)
  SGPTR@.Q_LEN_VECTOR := SGPTR@.Q_LEN_VECTOR +
   ((CLK - SGPTR@.Q_LEN_TIME) * SGPTR@.Q_LENGTH);
  SGPTR@.Q_LENGTH := SGPTR@.Q_LENGTH + 1;
  IF SGPTR@.Q_LENGTH > SGPTR@.MAX_Q_LENGTH THEN
     SGPTR@.MAX_Q_LENGTH := SGPTR@.Q_LENGTH;
  SGPTR@.Q_LEN_TIME := CLK;

END; (* ELSE Q EXISTS *)

(*-------------------------------------------------------*)


(*-------------------------------------------------------*)
(*-------------------------------------------------------*)

BEGIN (* PROCEDURE ARRIVE_AT_SG *)

  (* SEE IF JOB IS COMPLETED AND SEND IT OUT OF SYSTEM *)

  IF JPTR@.SERVING_EVENT = NIL THEN
     ARRIVECASE:= 1

  ELSE
    BEGIN (* JOB IS NOT COMPLETE, ATTATCH TO A SG *)

      (* FIND THE SG TO ATTATCH IT TO *)

      SGPTR:= FIRST_SGPTR;
      WHILE (SGPTR@.SERVER_GROUP <>
          JPTR@.SERVING_EVENT@.SERVER_GROUP_NO ) DO
        SGPTR:= SGPTR@.NEXT_SERVER_GROUP;
```

```
        (* IF THE SG HAS A Q THEN INSERT JCB IN THE Q *)
        IF SGPTR@.FIRST_IN_Q <>  NIL THEN
           ARRIVECASE:= 2
        ELSE  (* THERE IS NO Q AT THE SG *)
          BEGIN (* NO Q *)
            (* LOOK FOR AVAILABLE SERVER *)
            AVAIL_SERVER:= FALSE;
            SPTR:= SGPTR@.FIRST_SERVER;
            WHILE ((AVAIL_SERVER = FALSE) AND
                   (SPTR <> NIL)) DO
                IF SPTR@.BUSY = TRUE THEN
                  SPTR:= SPTR@.NEXT_SERVER
                ELSE
                  AVAIL_SERVER:= TRUE;
            (* IF THERE IS AN AVAILABLE SERVER ATTACH
               JCB TO THE SERVER.  OTHERWISE ATTACH TO
               FIRST IN Q *)
            IF AVAIL_SERVER THEN
              ARRIVECASE:= 3
            ELSE
              ARRIVECASE:= 4;
          END; (* NO Q *)
      END; (*JCB IS NOT COMPLETE *)
    CASE ARRIVECASE OF
      1:  JOB_COMPLETED;
      2:  INSERT_IN_SG_Q;
      3:  ATTACH_JOB_TO_SERVER;
      4:  ATTACH_FIRST_IN_Q;
    END; (* CASE ARRIVECASE *)
  END; (* PROCEDURE ARRIVE_AT_SG*)
```

```
(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)

BEGIN    (* EXECUTE AND TABULATE MAIN PROCEDURE  *)


CLK := 0;
CREATE_SERVER_GROUPS;
EXITQPTR := NIL;

     PAGE(OUTFILE);

(* GO INTO MAIN PROCESSING LOOP
   CONTINUE UNTIL THE MASTER EVENT Q IS
   EMPTY (ALL JOBS PROCESSED) OR UNTIL
   THE CLOCK TIME IS UP. *)

WHILE MASTERQPTR <> NIL DO

  BEGIN
    DEPART_FROM_SG;
    IF SGPTR@.NEXT_EVENT_TIME <> 0 THEN
       INSERT_IN_QUEUE (SGPTR, MASTERQPTR);
    ARRIVE_AT_SG;
  END;

  SIMNUM:= SAVE_SIMNUM;
  HIGHEST_JOB_TYPE := NEXT_JT_NUM - 1;
  STATS_FOR_JOBS;
  STATS_FOR_JOB_TYPES;
  STATS_FOR_SERVER_GROUPS;
  PAGE (OUTFILE);
  IF PRINT THEN
    PRINT_JS (EXITQPTR);

END; (* PROCEDURE EXECUTE AND TABULATE *)


(*-------------------------------------------------------------*)
(*          END EXECUTE AND TABULATE MODULE                    *)
(*-------------------------------------------------------------*)

(*-------------------------------------------------------------*)
(*-------------------------------------------------------------*)
(*                    CPMT MAIN DRIVER                         *)
(*-------------------------------------------------------------*)
```

```
(*-------------------------------------------------------------*)

BEGIN  (* MAIN DRIVER PROCEDURE *)

  OPEN (RECFILE,
        HISTORY := UNKNOWN,
        ORGANIZATION := INDEXED,
        ACCESS_METHOD := KEYED);

  OPEN (OUTFILE, 'OUTFILE.DAT', HISTORY:= NEW);
  REWRITE (OUTFILE);

  EXIT_MAIN := FALSE;
  REPEAT
    CLEAR_SCREEN;
    PRINTLN_MSG (OUTPUT, MESSAGES, MAIN_MENU);
    READLN (MAIN_OPT);
    IF MAIN_OPT <= 8 THEN
      CASE MAIN_OPT OF

        1: UPDATE_MENU;
        2: PRINT_DATA_BASE;


        3: BEGIN
             WRITELN ('ENTER NUMBER OF SIMULATION MODEL TO CHECK');
             READLN (SIMNUM);
             CHECK_SIM_SPECS (SIMNUM, SIMCHECK);
             IF SIMCHECK THEN
               WRITELN ('SIMULATION SPECIFICATIONS CHECK')
             ELSE
               WRITELN ('SIMULATION SPECIFICATIONS DID NOT CHECK. ',
                        'ERROR MESSAGES IN FILE OUTFILE.DAT');
             WRITELN;
             WRITELN ('ENTER ANY CHAR TO RETURN TO MAIN MENU ');
             READLN (ANS);
           END;


           (* EXECUTE A SIMULATION  *)

        4: BEGIN
             PRINTLN_MSG (OUTPUT, MESSAGES, SIMPAR);
             READLN (SIMNUM);
             PRINTLN_MSG (OUTPUT, MESSAGES, NUM_MSG);
             READLN (NUMJOBS);
```

```
                PRINTLN_MSG (OUTPUT, MESSAGES, SEEDPAR);
                READLN (SEED);

                SAVE_SIMNUM:= SIMNUM;
                WRITELN (OUTFILE, 'SIMULATION NUMBER IS ',
                         SIMNUM);
                CHECK_SIM_SPECS (SIMNUM, SIMCHECK);
                IF NOT SIMCHECK THEN
                  BEGIN
                    WRITELN ('SIMULATION SPECIFICATIONS DO NOT CHECK ');
                    WRITELN ('ERROR MESSAGES IN FILE OUTFILE.DAT');
                    WRITELN ('SIMULATION MODEL NOT EXECUTED');
                  END
                ELSE
                  BEGIN (* ELSE RUN THE SIMULTATION *)

                    WRITELN (OUTFILE, 'SEED IS ', SEED);
                    WRITELN (OUTFILE, 'NUMBER JOBS RUN IS ', NUMJOBS);

                    SIMNUM:= SAVE_SIMNUM;
                    CREATE_JOB_STREAM (ARRIVEQPTR, NUMJOBS);
                    EXECUTE_AND_TABULATE (ARRIVEQPTR);
                    WRITELN ('SIMULATION MODEL EXECUTED.  OUTPUT STATISTICS ',
                             'IN FILE OUTFILE.DAT ');
                  END; (* ELSE RUN THE SIMULATION *)

                WRITELN ('ENTER ANY CHAR TO RETURN TO MAIN MENU');
                READLN (ANS);

              END;

            8: EXIT_MAIN := TRUE;

            END; (* CASE MAIN_OPT OF *)

          UNTIL EXIT_MAIN;
          CLOSE (RECFILE);
          CLOSE (OUTFILE);

        END. (* MAIN DRIVER PROCEDURE *)

        (*-------------------------------------------------*)
        (*                END CPMT PROGRAM                 *)
        (*-------------------------------------------------*)
```

LIST OF REFERENCES

1. Morris, Michael F. and Roth, Paul F. *Computer Performance Evaluation* Van Nostrand Reinhold Company, New York, 1982.

BIBLIOGRAPHY

Borovits, Israel and Neumann, Seev, Computer Systems Evaluation, D.C. Heath and Company, 1979.

Emshoff, James E. and Sisson, Roger L., Design and Use of Computer Simulation Models, Macmillan Publishing Co., 1970.

Jacoby, Samuel L. S. and Kowalik, Janusz S. Mathematical Modeling With Computers, Prentice-Hall, Inc., 1980.

Sauer, Charles H. and Chandy, K. Mani, Computer Sytems Performance Modeling, Prentice-Hall, Inc., 1981.

Shannon, Robert E., Systems Simulation, Prentice-Hall, Inc., 1978.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943	2
3.	LtCol Alan A. Ross Code 52RS Naval Postgraduate School Monterey, California 93943	2
4.	Norman R. Lyons Code 54LB Naval Postgraduate School Monterey, California 93943	1
5.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
6.	LT Karen A. Pagel Navy Dept. OP16 Washington, D.C. 20350	1

END

FILMED

7-85

DTIC